Ricardo Jorge de Aragão Vaz Alves

Licenciatura em Engenharia Informática

Database Repairs With Answer Set Programming

Dissertação para obtenção do Grau de Mestre em Engenharia Informática

Orientador: Prof. Doutor João Alexandre C. Pinheiro Leite, Prof. Auxiliar, Universidade Nova de Lisboa

Júri:

Presidente: Prof. Doutor Adriano Martins Lopes

Arguentes: Prof[a]. Doutora Maria Inês C. de Campos Lynce Faria

Vogais: Prof. Doutor João Alexandre C. Pinheiro Leite

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Novembro, 2011

Database Repairs With Answer Set Programming

To my parents

Abstract

Integrity constraints play an important part in database design. They are what allow databases to store accurate information, since they impose some properties that must always hold. However, none of the existing Database Management Systems allows the specification of new integrity constraints if the information stored is already violating these new integrity constraints.

In this dissertation, we developed *DRSys*, an application that allows the user to specify integrity constraints that he wishes to enforce in the database. If the database becomes inconsistent with respect to such integrity constraints, *DRSys* returns to the user possible ways to restore consistency, by inserting or deleting tuples into/from the original database, creating a new consistent database, a database repair. Also, since we are dealing with databases, we want to change as little information as possible, so *DRSys* offers the user two distinct minimality criteria when repairing the database: minimality under set inclusion or minimality under cardinality of operations.

We approached the database repairing problem by using the capacity of problem solving offered by Answer Set Programming (ASP), which benefits from the simple specification of problems, and the existence of "Solvers" that solve those problems in an efficient manner.

DRSys is a database repair application that was built on top of the database management system PostgreSQL. Furthermore, we developed a graphical user interface, to aid the user in the whole process of defining new integrity constraints and in the process of database repairing.

We evaluate the performance and scalability of *DRSys*, by presenting several tests in different situations, exploring particular features of it as well, in order to understand the scalability of *DRSys*.

Keywords: answer set programming, relational databases, integrity constraints, inconsistency, repairs, minimality.

Resumo

Restrições de integridade desempenham um papel importantíssimo na implementação e desenho de uma base de dados. Elas permitem que as bases de dados contenham informação mais "exacta", visto que impõem certas propriedades que têm que ser respeitadas. Porém, nenhum dos actuais Sistemas de Gestão de Bases de Dados permite a especificação de novas restrições de integridade se a informação guardada na base de dados já estiver a violar estas novas restrições.

Nesta dissertação, desenvolvemos *DRSys*, uma aplicação que permite ao utilizador a especificação de restrições de integridade que devem ser impostas na base de dados. Se esta ficar inconsistente, de acordo com as restrições de integridade, *DRSys* retorna ao utilizador possiveis formas de restaurar consistência, através da inserção e/ou remoção de tuplos da base de dados original, gerando uma nova base de dados consistente, uma reparação, . Visto que estamos a lidar com bases de dados, queremos alterar o mínimo de informação possível. Desta forma, *DRSys* oferece ao utilizador dois critérios distintos de minimalidade: minimalidade sob inclusão de conjuntos ou minimalidade sob cardinalidade de operações (inserções e/ou remoções).

Abordámos o problema da reparação de bases de dados usando a capacidade de resolução de problemas oferecida pela Programação Por Conjuntos de Resposta, que benificia da simples espeficicação de problemas, e da existência de "Solvers" que resolvem estes problemas de uma maneira eficiente.

DRSys é uma aplicação de reparação de bases de dados que foi construído em cima do sistema de gestão de bases de dados PostgreSQL. Desenvolvemos também uma interface gráfica orientada ao utilizador, de forma a ajudar o utilizador no processo da especificação de restrições de integridade e reparação da base de dados.

Avaliamos a performance e escalabilidade de *DRSys*, mostrando diversos testes em diversas situações, explorando funcionalidades particulares de *DRSys*, de forma a compreender melhor a sua escalabilidade.

Palavras-chave: programação por conjuntos de resposta, bases de dados relacionais, restrições de integridade, inconsistência, reparação, minimalidade.

Contents

List of Figures

List of Tables

1

Introduction

Has everyone noticed that all the letters of the word database are typed with the left hand? Now the layout of the QWERTY typewriter keyboard was designed, among other things, to facilitate the even use of both hands. It follows, therefore, that writing about databases is not only unnatural, but a lot harder than it appears.

-Anonymous

Information, nowadays, is one of the most powerful assets available to human kind. Every company must store information about its employees, every bank must hold records for its customers and accounts and every school needs to keep data about its students. Information must be stored somewhere, and it must be easily and quickly accessed and retrieved. This growing need to deal with information led to the creation of database management systems. A database management system (DBMS) is a collection of interrelated data and a set of programs to access that data. The collection of data, usually referred to as a database, contains the information relevant to an enterprise. The main goal of a DBMS is to provide a way to store and retrieve information that is both convenient and efficient [SKS05].

From the earliest days of computers, storing and manipulating data has been a major application focus. The first general-purpose DBMS, designed by Charles Bachman at General Electric in the early 1960s[Bac73], was called the Integrated Data Store, and introduced the capability to link records in different files, later known as the "Network Data Model".

At this point, there was one "obvious" way to store the information, namely using the operating system's (OS) file system. But was it efficient? Imagine a bank, and the

information it has to store about its customers, accounts and employees, and so on. With the OS file system, we would need to create different files, one for each piece of information we wanted to store, i.e., one file to store information about the customer, another one to store information about the customer's account, and many more. Now suppose that, at first, we only wanted to consult the customer's name and the respective account. A programmer would need to write a program to answer this query. Then, if we realized we needed more information, another program would have had to be written. Furthermore, if we we thought about adding some more information related to the customer, we would need to re-write the customer's file as well. As we can see, this is not a very efficient manner to deal with this problem, which can also lead to some major problems, such as:

- **Data redundancy and inconsistency** - Information may be duplicated in different files, which may lead to inconsistency;
- **Difficulty in accessing data** - If further information is needed from the database, we would have needed to re-write the program, since it did not supported this new query;
- **Integrity Problems** - Data must, most of the time, satisfy certain types of consistency constraints. This was achieved by explicitly adding them in the code. If *a posteriori* we wanted to add a new one, we may have had to re-write, once more, the entire code.

In 1970, Edgar Codd proposed a new data representation framework, called the relational data model [Cod70]. This proved to be a watershed in the development of database systems [RG03]. Many of the database systems today are based on this concept. It clearly revolutionized the DBMS industry. A database using the relational model (or simply relational database) is, in a very summarized way, a set of relations with rows of tuples (and their values) that store information, and that are related to each other.

The way of storing information also evolved. The file system was not directly used any more. A new layer was developed to store and manage information more easily, providing the user a direct, easy and efficient way to implement a given data model, using specialized algorithms/structures (hidden from most users). This layer also provided means of ensuring correctness of the data.

The user was then allowed to define the database structure and required mechanisms to manipulate the stored information. But storing and manipulating are two distinct concepts. Therefore, some "language" had to be created to express what we want to store, how we want to store it, and what information we want to get. For this purpose, specific languages were created, the most widely used being SQL, which stands for Structured Query Language.

As the needs for storing information grew, the needs to, in a way, "filter" information, had to be developed as well. It is often the case that the user wants to store only some

specific information, restricting its possible values. Consider a bank enterprise. Surely, they don't want to allow a customer to have two different birth dates. For this purpose, integrity constraints were introduced. They represent an important source of information about the real world. They are used to define constraints on data. They also have a wide applicability in several contexts, such as semantic query optimization, cooperative query answering, database integration and view updates [GGZ03]. However, imposing these constraints can lead to costly computation. In this content, the most widely known integrity constraints in the literature are[SKS05]:

- **Functional Dependencies** - They state that for some set of values of a set of attributes of a relation, there can only be one, and only one, set of values of another set of attributes of the same relation;
- **Inclusion Dependencies** - They state that the set of values of a set of attributes from a relation, must exist as the set of values of a set of attributes from another relation (possibly the same);
- **Denial Constraints** - They state that a certain general property must hold in the database. The two most common denial constraints are:
  - **Domain Constraints** - They state that the value of an attribute must belong to a specific domain (a subset of the original domain of the attribute).
  - **Check Constraint** - They state that the value of an attribute must obey some condition.

The purpose of a database is to represent a "world" through a set of facts. Integrity constraints are used to restrict specific representations of the "world". The "world" we wish to represent keeps changing, therefore, it is not that difficult to see that we may have to implement further integrity constraints. But, by doing so, our existing data may become inconsistent.

Consider the relation in Table 1.1. It contains information about the employee's name and their sources of income, which may be in the form of a pension or a salary. In this case, consider John as being a doctor in a Hospital, for which he receives a salary, and he also receives a pension from having been the Minister of Health.

Name	*Money*	*From*
John	*123*	*Salary*
John	*456*	*Pension*
Mary	*789*	*Salary*

Table 1.1: Employees table

At a certain point, a new law was approved, stating that a person can only have one source of income, i.e., can only receive either a salary or a pension, but not both. We need

to enforce this new integrity constraint in our database. The integrity constraint would state that each person (value of attribute *Name*) can only be associated with one form of payment (value of attribute *Form*). It is clear that this instance violates the integrity constraint, since John has both sources of income.

How should we proceed? On the one hand, we need to enforce the integrity constraint and restore consistency, while on the other hand, we do not want to throw away the entire database.

There are two main approaches to deal with this problem. One is called *Consistent Query Answering*, technique introduced in [ABC99]. Here, the objective is to keep the database inconsistent and, despite it, to try to give consistent query answers. By consistent query answers, we mean the values of attributes that are not violating the defined integrity constraints. Going back to Table 1.1, suppose we wanted to know who has, as source of income, a salary. In that situation, we would only get *Mary*. *John* would never be returned, since it is inconsistent with respect to the integrity constraint.

The other approach is called *Database Repairing*, having been introduced in [ABC99], and further developed in [ABC00]. This technique deals with the inconsistency created by the introduction of a new integrity constraint by repairing the database, generating a new database that, when replacing the previous inconsistent one, restores consistency. Going back to Table 1.1, using *Database Repairing*, two possible ways to repair the database would be to delete either $\langle John, 123, Salary\rangle$ or $\langle John, 456, Salary\rangle$, generating the following repairs:

Name	*Money*	*From*
John	*456*	*Pension*
Mary	*789*	*Salary*

Name	*Money*	*From*
John	*123*	*Salary*
Mary	*789*	*Salary*

Table 1.2: Employees Repair

We could also delete all tuples from the relation, being this still a possible repair. We could also delete one tuple, where the name John appears as value of the attribute *Name*, and add another tuple, like *Employee(Richard, 456,Pension)*, being the resulting instance still a possible repair. Actually, there may be infinite repairs (if we consider the possible insertion of new tuples). In order to cover this problem, the new database instance should be minimally different from the original one, according to some notion of minimality. Finding repairs is, in general, a problem that lies between the *NP-hard* and the Σ_2^p complexity cases [CM05].

Consistent Query Answering and Database Repair also differ in the situations on which they are more appropriately applicable. Consistent query answering is very useful when we do not have permissions to alter the database, generally in a distributed database system. Database repairing is very useful otherwise, generally in a centralized database system, where the user usually has permissions to change the data.

In this dissertation, we focused on the database repair problem. By using database repairing, we want the user to be able to express new integrity constraints in an already existing database. If the database becomes inconsistent with respect to the new integrity constraints, we want to repair the database, by deleting or adding tuples, such that, the final outcome, the repaired database, is consistent with respect to the integrity constraints.

Given the computational complexity of such technique, we want to explore this problem by using Answer Set Programming (ASP) [GL88, GL91]. Answer Set Programming is a form of declarative programming oriented towards difficult, primarily *NP-hard* problems. Its main idea is to represent a given computational problem by an answer set program (a kind of logic program) whose answer sets (solutions/models) correspond to solutions of the real problem[Lif02]. Furthermore, the existence of "Answer Set Solvers" (tools developed to compute answer set programs) allow us to solve such problems in an efficient manner.

ASP is becoming a growing tool, and getting more and more powerful. Nowadays, it has already been applied to several areas of science and technology, such as *Automated Product Configuration*(Tiihonen *et al.* 2003), *Decision Support For the Space Shuttle*(Nogueira *et al.* 2001) and *Inferring Phylogenetic Tree*(Brooks *et al.* 2007).

Despite ASP being oriented towards solving *NP-hard* problems, some solvers allow the use of optimization statements that allow the solving of problems with higher complexity, namely some in the Σ_2^p complexity class. Therefore, the use of ASP is adequate to address the database repairing problem.

In this dissertation, we implemented *DRSys*[1], a practical solution that allows the user to freely define new integrity constraints and, if necessary, computes repairs using *Answer Set Programming*. In *DRSys*, we allow the user to define new integrity constraints that he wishes to enforce in the database, and, if they lead the database to an inconsistent state, *DRSys* computes the possible database repairs according to the minimality criteria chosen by the user - minimality under set inclusion or minimality under cardinality of operations. Furthermore, *DRSys* also allows the user to input some knowledge that can greatly increase its performance, by limiting the maximum number of tuples that can be deleted, by limiting the maximum number of tuples that can be inserted, by limiting the overall number of operations that can be performed and by restraining some specific tuples from being deleted. Also, *DRSys* provides mechanisms to automatically create some integrity constraints directly into answer set programming, allowing the user with no knowledge on answer set programming to use *DRSys*. *DRSys* also allows the specification of integrity constraints directly in SQL.

The main contributions and achievements described in this dissertation are:

- The definition of a transformation function, that maps the database repair problem into a logic program;

[1] *DRSys* is available online and can be downloaded at http://sourceforge.net/projects/drsys/files/drsys1.0/

- The implementation of a practical solution of the database repair problem, as well as the development of a graphical user interface, allowing the interaction between the user and the application;
- The testing of the performance of our application, by discussing the results in different scenarios;
- The study of the related work and comparison of our work with others presented in the database repair problem literature.

The plan of this document is as follows: in Chapter 2, we recall First Order Logic, the Relational Model and Answer Set Programming. In Chapter 3, we formally introduce the problem of database repair. In Chapter 4, we discuss related work. In Chapter 5, we discuss our approach, database repairing using answer set programming. In Chapter 6, we present the interface and architecture of the developed application. In Chapter 7, we present experimental results of our application, comparing our application with others developed in the literature. Finally, in Chapter 8, we present our conclusions and future work.

2

Preliminaries

2.1 First Order Logic

First Order Logic (FOL) is a formal logical system used in mathematics, philosophy, linguistics and computer science. In this section, we recall the main concepts and terminology of FOL, by going through its syntax and semantics.

2.1.1 FOL - Syntax

Every first order language has some usual notations, being them the following[Fit96]:

- **Propositional Connectives** are the same as in propositional logic, where we have the usual symbols $\neg, \subset, \supset$ and the propositional constants $\top, \bot$.
- **Quantifiers**
  - $\forall$ (for all, the universal quantifier)
  - $\exists$ (exists, the existential quantifier)
- **Punctuation** '(', ')', ','
- **Variables** $x, y, z...$ They may also be used together with subscripts, as $x_1, x_2, y_1, y_2, z_1, z_2...$

Definition 2.1 (First-Order Language)**.** *A first-order language is determined by specifying:*

1. *A finite set R of predicate symbols, $p, q, ...$, each of which having a positive integer associated with it. If $p \in R$ has the integer n associated with it, we say p is an n-place relation symbol.*

Also, consider the following special 2-placed relation symbols: $=$ and $\neq$, where the first stands for equality and the second for inequality, such that $=\in R$ and $\neq\in R$.

2. *A finite set F of function symbols, $f, g, h, ...$, each of which having a positive integer associated with it. If $f \in F$ has the integer n associated with it, we say f is an n-place function symbol;*

3. *A finite set C of constant symbols, $a, b, c, ...$ They may also be used together with subscripts, as $a_1, a_2, b_1, b_2, c_1, c_2, ...$*

We use the notation $L(R, F, C)$ for the first-order language determined by R, F and C. Having specified the basic elements of the syntax, the alphabet, we go on to more complex constructions.

Definition 2.2 (Term)**.** *The family of terms of $L(R, F, C)$ is the smallest set meeting the conditions:*

- *Any variable is a term of $L(R, F, C)$;*
- *Any constant symbol (member of C) is a term of $L(R, F, C)$;*
- *If f is a n-place function symbol (member of F), and $t_1, ..., t_n$ are terms of $L(R, F, C)$, then $f(t_1, ..., t_n)$ is a term of $L(R, F, C)$.*

Definition 2.3 (Closed Term)**.** *A term is closed if it contains no variables.*

Definition 2.4 (Atomic Formula)**.** *An atomic formula of $L(R, F, C)$ is any expression of the form $p(t_1, ..., t_n)$, where p is an n-place relation symbol (member of R) and $t_1, ..., t_n$ are terms of $L(R, F, C)$; also $\top$ and $\bot$ are taken to be atomic formulas of $L(R, F, C)$. Expressions of the form $= (t_1, t_2)$ (usually written $t_1 = t_2$) and , $\neq (t_1, t_2)$ (usually written $t_1 \neq t_2$) where t_1 and t_2 are terms, are also taken to be atomic formulas of $L(R, F, C)$.*

Definition 2.5 (Formula)**.** *The family of formulas of $L(R, F, C)$ is the smallest set meeting the conditions:*

- *Any atomic formula of $L(R, F, C)$ is a formula of $L(R, F, C)$;*
- *If A is a formula of $L(R, F, C)$, so is $\neg A$;*
- *If A and B are formulas, $A \supset B$ is also a formula of $L(R, F, C)$;*
- *If A is a formula of $L(R, F, C)$ and x is a variable, then $(\forall x)A$ and $(\exists x)A$ are formulas of $L(R, F, C)$.*

Note: *There are also some other connectives in a first order language, which are only used as short cuts, since they can be expressed using the previously defined connectives. We describe them next:*

- *If A and B are formulas, then $A \subset B$ is also a formula of $L(R, F, C)$, where $A \subset B = B \supset A$;*
- *If A and B are formulas, then $A \vee B$ is also a formula of $L(R, F, C)$, where $A \vee B = \neg A \supset B$;*
- *If A and B are formulas, then $A \wedge B$ is also a formula of $L(R, F, C)$, where $A \wedge B = \neg(A \supset \neg B)$.*

Definition 2.6 (Theory)**.** *A theory is a set of formulas of $L(R, F, C)$.*

We have now seen the syntax of FOL. Now, we can proceed to the semantics.

2.1.2 FOL - Semantics

The semantics of a first-order language gives meaning to a formula. In order to do so, we define some important notions.

Definition 2.7 (Interpretation)**.** *An interpretation (structure) for the first order language $L(R, F, C)$ is a pair $M = \langle D, I \rangle$, where:*

- *D is a non-empty set, called the domain of M;*
- *I is a mapping, called the interpretation function, that associates:*
  - *to every constant symbol $c \in C$ some member $c^I \in D$;*
  - *to every n-place function symbol $f \in F$ some n-ary function $f^I : D^n \to D$;*
  - *to every n-place relation symbol $p \in R$ some n-ary relation $p^I \subseteq D^n$;*

Definition 2.8 (Assignment)**.** *An assignment in an interpretation $M = \langle D, I \rangle$ is a mapping A from the set of variables to the set D. We denote the image of the variable ν under an assignment A by ν^A.*

Suppose we have an interpretation and we have an assignment. We have now enough information to calculate values for arbitrary terms.

Definition 2.9. *Let $M = \langle D, I \rangle$ be an interpretation for the language $L(R, F, C)$, and let A be an assignment in this interpretation. To each term t of $L(R, F, C)$, we associate a value $t^{I,A}$ in D as follows:*

1. *For a constant symbol, c, $c^{I,A} = c^I$;*
2. *For a variable ν, $\nu^{I,A} = \nu^A$;*
3. *For a function symbol f, $[f(t_1, ..., t_n)]^{I,A} = f^I(t_1^{I,A}, ..., t_n^{I,A})$.*

Next, we want to associate a truth value with each formula of the language, with respect to an interpretation and an assignment. Before introducing the truth valuation function, let us introduce the notion of an *x-variant*.

Definition 2.10 (x-variant). *Let x be a variable. The assignments A and B in the interpretation M are x-variants if A and B assign the same values to every variable, except possibly x.*

Now we can define our truth valuation function like the following:

Definition 2.11 (Truth Valuation). *Let $M = \langle D, I\rangle$ be an interpretation for the language $L(R, F, C)$ and let A be an assignment in this interpretation. To each formula Φ of $L(R, F, C)$, we associate a truth value $\Phi^{I,A}$ (1 or 0), as follows:*

1. *For the atomic cases**:*
   - $[p(t_1, ..., t_n)]^{I,A} = 1 \iff \langle t_1^{I,A}, ..., t_n^{I,A}\rangle \in p^I$;
   - $\top^{I,A} = 1$;
   - $\bot^{I,A} = 0$;
   - $[t_1 = t_2]^{I,A} = \begin{cases} 1 & \text{if} \quad [t_1]^{I,A} = [t_2]^{I,A} \\ 0 & \text{if} \quad otherwise \end{cases}$
   - $[t_1 \neq t_2]^{I,A} = \begin{cases} 0 & \text{if} \quad [t_1]^{I,A} = [t_2]^{I,A} \\ 1 & \text{if} \quad otherwise \end{cases}$
2. $[\neg X]^{I,A} = 1 - [X^{I,A}]$;
3. $[X \supset Y]^{I,A} = 1 \iff X^{I,A} = 0$ *or* $Y^{I,A} = 1$;
4. $[(\forall x)\Phi]^{I,A} = 1 \iff \Phi^{I,B} = 1$ *for every assignment B in M that is an x-variant of A;*
5. $[(\exists x)\Phi]^{I,A} = 1 \iff \Phi^{I,B} = 1$ *for some assignment B in M that is an x-variant of A;*

Definition 2.12. *A formula Φ of $L(R, F, C)$ is true in the interpretation $M = \langle D, I\rangle$, denoted $M \models \Phi$, if $\Phi^{I,A} = 1$ for all assignments A. A theory S is satisfiable in $M = \langle D, I\rangle$, denoted $M \models S$, if there is some assignment A such that $\Phi^{I,A} = 1$ for all $\Phi \in S$.*

Definition 2.13 (Model). *Given an interpretation $M = \langle D, I\rangle$ and a theory S, M is a model of S if S is satisfiable in M.*

2.2 Relational Databases

In this section, we formally introduce the relational model, by introducing the structure of relational databases. Also, we formally present integrity constraints, focusing on the most widely used in the database literature: key constraints, functional dependencies, inclusion dependencies, and denial constraints, with emphasis on check constraints and domain constraints. Furthermore, we introduce an alternative representation of the database, by mapping the database into a first order logic theory.

*Since we introduced first order logic with equality and inequality, to be completely formal, we should have introduced the predicates $>, <, \geq, \leq$, for integers, and for Strings. As a question of simplicity, we opted not to introduce them.

2.2.1 Relational Model

Codd proposed the relational data model back in 1970. This model revolutionized the database field and, nowadays, is the primary data model for commercial data-processing applications, mainly because of its simplicity, elegance and expressiveness, easing the job of the programmer, compared to earlier data models.

The main construct for representing data in the relational model is a relation. A relation can be seen as a table, where the columns represent the attributes, and the rows represent the information stored. For each attribute, there is a set of permitted values called the domain of the attribute. Since relations are essentially tables, we can use the terms **relation** and **tuple** in place of the terms **table** and **row** respectively.

A relation consists of a relation schema and a name. A relation schema is the logical design of a relation. It contains the name and domain of each attribute, i.e., the set of permitted values the attribute can take. The arity of a relation is the number of attributes in it.

Let us define $\mathcal{ATT}$ as the set of all names for the attributes of all relations in a database. By $\mathcal{N}$, we mean the set of all names of relations in a database.

Throughout this section, we use $\kappa, \kappa_0, \kappa_1, ..., \kappa_n$ and $\varepsilon, \varepsilon_0, \varepsilon_1, ..., \varepsilon_m$ to denote names of attributes, i.e., $\kappa_i \in \mathcal{ATT}$ and $\varepsilon_j \in \mathcal{ATT}$.

Let us now formally introduce the definition of a relation schema.

Definition 2.14 (Relation Schema)**.** *A relation schema is a sequence of pairs of the form:*

$$S = \langle\langle \kappa_0, D_0\rangle, \langle \kappa_1, D_1\rangle, ..., \langle \kappa_n, D_n\rangle\rangle$$

where $\kappa_0, \kappa_1, ..., \kappa_n$ are names of attributes such that, whenever $i \neq j$, $\kappa_i \neq \kappa_j$, and $D_0, D_1, ...,$ D_n are domains of the attributes $\kappa_0, \kappa_1, ..., \kappa_n$ respectively.

Intuitively, a relation schema is a sequence of pairs of names of attributes and their corresponding domains.

An important feature to acquire from the previous definition is that there cannot be two distinct attributes with the same name belonging to the same relation schema.

It is important to establish the correct domain for the attributes. For instance, if we have an attribute with name *Birth Date*, used to represent the birth dates of a customer in a bank, we want the domain of that attribute to be of the Date type – we don't want to allow some birth date to be an arbitrary string.

We can now define what a relation is.

Definition 2.15 (Relation)**.** *A relation is a pair $\langle N, S\rangle$, where N stands for the name of the relation, such that $N \in \mathcal{N}$, and $S = \langle\langle \kappa_0, D_0\rangle, \langle \kappa_1, D_1\rangle, ..., \langle \kappa_n, D_n\rangle\rangle$ stands for the relation schema.*

As we can see in the relation depicted in Figure 2.1, the attributes of this relation are *Id*, *Name* and *Birth Date*, and the arity of the *Client* relation is three. Let D_1, domain of

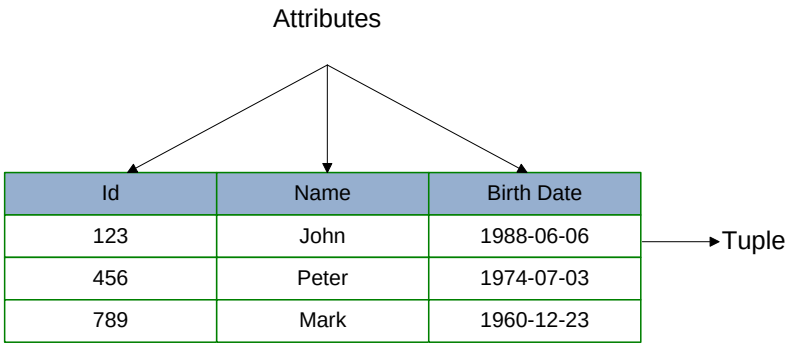

Figure 2.1: Client relation

the *Id* attribute, define the set of all identification numbers possible, D_2, domain of the *Name* attribute, the set of all possible names (or strings), and D_3, domain of the *Birth Date* attribute, the set of all dates. Then, any tuple of the *Client* relation must consist of a 3-tuple $\langle v_1, v_2, v_3 \rangle$ where v_1 is an identification number from the domain D_1, v_2 is a name from the domain D_2 and v_3 is a birth date from the domain D_3. In order to keep track of the information stored in a relation, the concept of relation instance was developed. It is a snapshot of a relation, at a given moment in time, i.e., it is a representation of a relation at a given moment in time. Therefore, and following the previous example, the content of the *Client* relation instance will contain only a subset of all possible combinations of every domain in the relation. The content of the relation instance of the *Client* relation, denoted here by C_{client}, is then a subset of the Cartesian product of the domains, i.e.:

$$C_{client} \subseteq D_1 \times D_2 \times D_3$$

We can now formally define what a relation instance, with respect to a specific relation, is.

Definition 2.16 (Relation Instance)**.** *Given a relation $R = \langle N, S \rangle$, such that N is a relation name, where $N \in \mathcal{N}$, and S is a relation schema, such that $S = \langle \langle \kappa_0\ D_0 \rangle, \langle \kappa_1, D_1 \rangle, ..., \langle \kappa_n, D_n \rangle \rangle$, a relation instance, with respect to relation R is a triple $\langle N, C, S \rangle$, where C, the content, is given by:*

$$C \subseteq D_0 \times D_1 \times D_2 ... \times D_{n-1} \times D_n$$

Notice that, as a consequence of the previous definition, there are no duplicates in the content of a relation instance, although, in many practical database management systems, every schema is extended with one additional internal attribute, hidden from the user, which is given an unique value for each tuple, hence allowing duplicate tuples if we only consider the original schema.

From now on, we shall assume that the schema of every relation from a database contains a special extra attribute, the $Rowid$ attribute, that uniquely identifies a tuple

within a database, allowing the distinction between seemingly duplicate tuples. Also, it will be represented as the first attribute on a relation schema, being denoted as the κ_0, attribute whose domain is the set of all positive integers, denoted by D_0. Also notice that this extra attribute is not included in the arity of a relation.

If we want to address the value of a particular attribute X of a particular tuple t, we can use the following notation $t[X]$. For instance, consider t being the first tuple of the relation depicted in Figure 2.1. Then, $t[Id] = 123$, $t[Name] = John$ and $t[BirthDate] = 1988-06-06$. Also, if X is a sequence of attributes, $t[X]$ represents the sequence of values of X. For instance, let $X = \langle Id, Name, BirthDate \rangle$. Then, $t[X] = \langle 123, John, 1988-06-06 \rangle$.

One domain value that is, by default, a member of all possible domains is the $NULL$ value, which means that the value is unknown or does not exist. However, it is often the case that the $NULL$ value is removed from some domain, enforcing the value to be known. In this dissertation, we did not consider the value $NULL$ to be present in the domain of any attribute. We made this choice since the ISO SQL implementation of the $NULL$ value is subject to criticism, debate and calls for change. There is no clear semantics associated with the $NULL$ value. It may represent an unknown value, or a missing value. Codd suggested that the $NULL$ value should be replace by two distinct $NULL$-type markers, standing for "Missing but Applicable" and "Missing but Inapplicable". Several other authors have also discussed the semantics of the $NULL$ value, but, ultimately, there is no global accepted meaning, thus the exclusion of it in our work.

A relational database is a collection of relations with distinct names. The relational database schema is a collection of schemas for the relations in the database and the relational database instance is the collection of instances for the relations in the database. Also, it is often the case that we want to verify certain properties in a database, so that we can have more accurate and consistent information stored, and also, to prevent accidental damage. For instance, we may not want the balance of a bank account to be below 0. Integrity constraints are a way to ensure these properties. They represent an important source of information about the real world[GGZ03]. They will be discussed in greater detail in the next section. Therefore, a database instance may also have some integrity constraints associated with it.

Definition 2.17 (Relational Database Instance)**.** *A relational database instance D is a pair of a set of relation instances, together with a set of integrity constraints IC, i.e.:*

$$D = \langle \{\langle N_1, C_1, S_1 \rangle, \langle N_2, C_2, S_2 \rangle, ..., \langle N_n, C_n, S_n \rangle\}, IC \rangle$$

From this point on, we shall use the terms database instance and database interchangeably, as well as relation instance and relation. If ambiguity may arrive, we shall use the exact terms.

It is important to have a way to know the names of the attributes of a relation, given the name of the relation. In order to do that, we defined the following function:

Definition 2.18 (Attributes Function). *Let $D = \langle I, IC\rangle$ be a database. Then, let $\mathcal{A}$ be a function such that, for every N such that $\langle N, S, C\rangle \in I$ and $S = \langle\langle\kappa_0, D_0\rangle, \langle\kappa_1, D_1\rangle, ..., \langle\kappa_n, D_n\rangle\rangle$,*

$$\mathcal{A}(N) = \langle\kappa_0, \kappa_1, ..., \kappa_n\rangle$$

Since the attributes of a relation must obey some order (follows from the definition of a relation schema), it is often the case where we want to know the index of a specific attribute, when given its name, from a given relation. This index will correspond to the position of the attribute in a relation. We defined the following function for that purpose:

Definition 2.19 (Attribute Index Function). *Let $D = \langle I, IC\rangle$ be a database. Then, let $\#$ be, for every relation $\langle N, C, S\rangle \in I$, where $S = \langle\langle\kappa_0, D_0\rangle, \langle\kappa_1, D_1\rangle, ..., \langle\kappa_n, D_n\rangle\rangle$, given an attribute name $\kappa \in \mathcal{ATT}$ and a relation name $N \in \mathcal{N}$, defined as follows:*

$$\#_\kappa^N = \begin{cases} undefined & \text{if} \quad \nexists_{i\in\{0,1,2,...,n\}}\kappa = \kappa_i \\ i & \text{if} \quad \exists_{i\in\{0,1,2,...,n\}}\kappa = \kappa_i \end{cases}$$

For a set of names of attributes $X = \{\varepsilon_1, \varepsilon_2, ..., \varepsilon_n\}$, let $\#_X^N$ be defined by:

$$\#_X^N = \{\#_\varepsilon^N \mid \varepsilon \in X\}$$

It is often the case that the arity of a specific relation must be known. For that purpose, we defined the following function:

Definition 2.20 (Arity Function). *Let $R = \langle N, C, S\rangle$ be a relation where $S = \langle\langle\kappa_0, D_0\rangle, \langle\kappa_1, D_1\rangle, ..., \langle\kappa_n, D_n\rangle\rangle$. Then, let $\mathcal{AR}$ be a function such that, given a relation name N, returns the arity of the relation with name N, defined as follows:*

$$\mathcal{AR}(N) = n$$

Up until now, we introduced the concepts of a relation and database as they are usually defined in the database literature. Since in this dissertation our objective is to use answer set programming, which is a kind of a logic program, we also show how a database can be represented as it is usually done in the logic programming literature. Therefore, it is well known that a relational database can be expressed as a first order logic theory. We now present a different representation of a database, where we view it as a set of logic atoms, where each atom represents a tuple from a specific relation, such that each atom is composed by a predicate name N, which is obtained from the name of the relation, and by a sequence of terms t_i, each of which corresponding to a value of an attribute of a tuple belonging to the content of an instance of a relation N. The name of the relation directly maps into the name of the predicate of the atom, and each value of an attribute of a tuple directly maps into a value of a term of an atom.

In order to define this mapping, we defined two distinct functions. In the first one, we define the mapping of a relation, and then, in the second one, we define the mapping

of a database.

Definition 2.21 (Relation Mapping). *Let $R = \langle N, C, S \rangle$ be a relation. Let β_p be an operator that maps a relation into a set of logic atoms, defined as:*

$$\beta_p(R) = \{N(t_0, t_1, t_2, ..., t_n) \mid \langle t_0, t_1, t_2, ..., t_n \rangle \in C\}$$

Now, we can define our database mapping as simply the union of all relations mappings.

Definition 2.22 (Database Mapping). *Let $D = \langle I, IC \rangle$, such that $I = \{R_1, R_2, ..., R_n\}$ be a database. Let β_t be an operator that maps the entire database instance into a set of logic atoms, by performing the following operation:*

$$\beta_t(D) = \bigcup_{R \in I} \beta_p(R)$$

From the previous definitions, we have defined a way to map the whole database into a set of logic atoms. We firstly defined the mapping of a single relation, and then, defined the mapping of the whole database simply as the union of the mappings of all relations belonging to the database.

Since a database is a pair of a set of relations and a set of integrity constraints, it is important to understand what these integrity constraints are, also because they are at the main core of this dissertation. We describe some of them next.

2.2.2 Integrity Constraints (IC's)

In a database, we represent a certain "world", and our data must respect some properties. These properties can be ensured by the use of integrity constraints. Integrity constraints are widely used in database management systems, and have a lot of power and expressiveness.

Integrity constraints are closed first-order $\mathcal{L}$-formulas, where $\mathcal{L}$ is a first-order language defined in the standard way. In the sequel, we denote relation symbols by $N, N_1, ...,$ N_m. We denote by $\bar{x}, \bar{y}, \bar{z}$, sequences of pairwise distinct variables with appropriate arity, such that x_i denotes the i^{th} component of $\bar{x}$. Also, we represent conjunctions of atomic formulas referring to built-in predicates by φ.

We now describe several integrity constraints classes.

2.2.2.1 Keys and Functional Dependencies

Within a relation, we must have a way to distinguish tuples. This is expressed in terms of the values of their attributes. We want the tuples to be uniquely identified by the set of values of a set of attributes. Therefore, the key integrity constraint was developed. In a relational database, they refer to a set of attributes of a relation for which it holds that no two distinct tuples have the same values for those attributes. Therefore, given a key

constraint, we have a set of one or more attributes that, taken collectively, allow us to uniquely identify a tuple in a relation.

Definition 2.23 (Keys)**.** *Given a relation name N, and a set of names of attributes A (the ones that form the key constraint), such that $\mathcal{AR}(N) = n$, the key constraint, with respect to N and A, expressed by $SK(N, A)$, is defined as follows:*

$$\forall_{\bar{x},\bar{y}} \neg \left[N(\bar{x}) \wedge N(\bar{y}) \wedge \bigwedge_{i \in \#_A^N} x_i = y_i \wedge \bigvee_{j \in \{0,1,2,3,\ldots,n\} \setminus \#_A^N} x_j \neq y_j \right] \tag{2.1}$$

Consider the *Customers* relation in Table 2.1 and the following integrity constraint: $SK(Customers, \{Id\})$

Customers

Id	*Name*
1	*John*
1	*Peter*
2	*Michael*

Table 2.1: Key constraint example

There is a clear violation of the key integrity constraint, since we have two distinct tuples in relation *Customers* with the same value for the attribute *Id*.

Keys can be seen as a special case of a more general type of integrity constraints: the functional dependencies. Given a relation R, a set of attributes X in R is said to functionally determine another set of attributes Y, also in R, if each X value is associated with precisely one Y value. This can also be expressed by $X \rightarrow Y$.

The difference between key constraints and functional dependencies, is that, whenever we have a key constraint, a set of attributes functionally determines all the attributes, while in the functional dependency, it may be the case where the attributes that are being functionally determined are not all attributes, but only some of them.

Definition 2.24 (Functional Dependencies)**.** *Given a relation name N, two sets of names of attributes A and B from relation N, such that $A \rightarrow B$, the functional dependency with respect to N, A and B, expressed by $FD(N, A, B)$, is defined as follows:*

$$\forall_{\bar{x},\bar{y}} \neg \left[N(\bar{x}) \wedge N(\bar{y}) \wedge \bigwedge_{i \in \#_A^N} x_i = y_i \wedge \bigvee_{j \in \#_B^N} x_j \neq y_j \right] \tag{2.2}$$

Since key constraints are a special case of the functional dependencies, we can define the first in terms of the second in the following way: given a relation N and the set A of attributes that form the candidate key, and recalling that the sequence of attributes of a

relation N, $\langle \kappa_0, \kappa_1, ..., \kappa_n \rangle$, is given by $\mathcal{A}(N)$:

$$SK(N, A) = FD(N, A, \{\kappa_0, \kappa_1, ..., \kappa_n\} \setminus A)$$

Intuitively, we are saying that a set of attributes functionally determines the whole rest of attributes existing the relation.

2.2.2.2 Inclusion Dependencies

Inclusion dependencies (INDs) state that the sequence of values of a sequence of attributes of the tuples in a relation, must exist as the sequence of values of a sequence of attributes of the tuples of another (possibly the same) relation.

Definition 2.25 (Inclusion Dependencies)**.** *Given two names of relations N_1 and N_2, two sequences of names of attributes $A = \langle \kappa_1, \kappa_2, ..., \kappa_k \rangle$ and $B = \langle \varepsilon_1, \varepsilon_2, ..., \varepsilon_k \rangle$ from relation N_1 and relation N_2 respectively, let the inclusion dependency, with respect to N_1, N_2, A, B, expressed by $IND(N_1, N_2, A, B)$, where the values of the attributes in A must exist as the values of the attributes in B, defined as:*

$$\forall_{\bar{x}} \exists_{\bar{y}} \left[\neg N_1(\bar{x}) \vee (N_2(\bar{y}) \wedge \bigwedge_{i=1}^{k} x_{\#_{\kappa_i}^{N_1}} = y_{\#_{\varepsilon_i}^{N_2}}) \right] \tag{2.3}$$

From the previous definition, we conclude that if a sequence of values for a sequence of attributes exists in relation N_1, the same sequence of values for another sequence of attributes must exist in relation N_2. If it doesn't exist in N_1, it may or may not exist in N_2.

This is often written as $N_1[Y_1] \subseteq N_2[Y_2]$ where Y_1 (respectively Y_2) is the sequence of values of the sequence of attributes of N_1 (respectively N_2) corresponding to the attributes that form the inclusion dependency [Cho07].

Accounts

Account Number	*Branch Name*
111	*Sete Rios*
222	*FCT*
333	*Coimbra*

Branches

Branch Name	*Branch City*
Sete Rios	*Lisboa*
FCT	*Almada*
Benfica	*Lisboa*
Aliados	*Porto*

Table 2.2: Relations of the example

Consider the following relations: *Accounts* and *Branches*, in Table 2.2. If we take into consideration the inclusion dependency, represented by *Accounts*[*Branch Name*] $\subseteq$ *Branches* [*Branch Name*], there is a clear violation of the integrity constraint, since *Coimbra* does not exist as the value of the attribute *Branch Name* in relation *Branches*.

2.2.2.3 Denial Constraints

Denial constraints (DCs) are a kind of integrity constraint that prevent a certain general property to hold in a database. We present two special cases of the denial constraints, the *check constraints* and the *domain constraints*. Afterwards, we formally introduce the definition of a denial constraint.

Check constraints are a kind of integrity constraint that restrict the domain of a specific attribute, by imposing some mathematical restriction over the value of that attribute, that must always be obeyed.

Definition 2.26 (Check Constraint)**.** *Given a relation name N, an attribute name κ, a mathematical operator $\theta \in \{>, <, \geq, \leq, =, \neq\}$ and a value V of the domain of κ, let a check constraint, with respect to N, κ, θ and V, expressed by $CC(N, \kappa, \theta, V)$, be defined as follows:*

$$\forall_{\bar{x}} \neg [N(\bar{x}) \wedge x_{\#_\kappa^N} \ \theta \ V] \tag{2.4}$$

Examples of check constraints may be: a person must be over 21 years old to be a costumer in a bank, a bank account balance must be greater than 100, and so on. Consider Table 2.3, representing the *Employee* relation, belonging to a database D:

Name	*Age*
John	*22*
Peter	*32*
Paul	*35*

Table 2.3: The Employee relation

Now consider the check constraint constraint: $CC(Employee, Age, <, 36)$. The relation would be inconsistent, since there are people younger than 36.

Domain constraints are a kind of integrity constraint that do not allow the value of a certain attribute to be outside of a user specified set of values. We restrict the domain Do of an attribute to a subset Do' of Do.

Definition 2.27 (Domain Constraints)**.** *Given a relation name N, an attribute name κ and a specific domain Do of the form $Do = \{val_1, val_2, ..., val_k\}$, let a domain constraint, with respect to N, κ and Do, expressed by $DoC(N, \kappa, Do)$, be defined as follows:*

$$\forall_{\bar{x}} \neg \left[N(\bar{x}) \wedge \bigwedge_{i=1}^{k} x_{\#_\kappa^N} \neq val_i \right] \tag{2.5}$$

Examples of domain constraints may be: the sex of a person must be either Male of Female; a person's civil state should be either single, married or divorced. Consider Table 2.4, where we have the *Movies* relation, where we specify informations about a movie.

Movie	*Genre*
M1	*Action*
M2	*Action*
M3	*Drama*
M4	*Romance*

Table 2.4: The Movies relation

Now consider the domain constraint: $DoC(Movies, Genre, \{Action, Drama, Comedy\})$. The relation would be inconsistent, since we have a genre *Romance*.

Both of the previous integrity constraints are a special case of a more general integrity constraint, the denial constraint.

Definition 2.28 (Denial Constraints)**.** *Given m names of relations, $N_1, N_2, ..., N_m$, where $\bar{x}_i$ is the sequence of variables of relation N_i, and some built in predicates $\varphi(\bar{x}_1, ..., \bar{x}_m)$ describing some general properties of the database that must never be true, let then a denial constraint be defined as follows:*

$$\forall_{\bar{x}_1,...,\bar{x}_m}.\neg[N_1(\bar{x}_1) \wedge ... \wedge N_m(\bar{x}_m) \wedge \varphi(\bar{x}_1, ..., \bar{x}_m)] \tag{2.6}$$

Integrity constraints are a powerful tool to express certain properties in a database. These properties must be guaranteed by the database, in order to have consistent data.

2.3 Answer Set Programming

Answer Set Programming is a form of declarative programming oriented towards difficult, primarily *NP-hard* problems, built on the foundation of logic programming with negation [Lif08]. As an outgrowth of research on the use of nonmonotonic reasoning in knowledge representation, it is particularly useful in knowledge-intensive applications. ASP is based on the stable model (answer set) semantics of logic programming[GL88].

In ASP, search problems are reduced to computing stable models, and answer set solvers - programs for generating stable models - are used to perform search. The use of answer set solvers for search was identified as a new programming paradigm in [MT98].

Throughout this chapter, we formally introduce the stable models, its syntax and afterwards, its semantics.

The most popular and widely used solvers are: smodels [smo, NS97], DLV [dlv, ELM+98] and more recently, clasp[cla, GKNS07].

2.3.1 Syntax

In logic programming, much of the terminology is borrowed from first order logic. The alphabet $\mathcal{A}$ of a logic programming language $\mathcal{L}$ is defined almost in the same way as the

first order logic alphabet. We shall introduce new symbols to the alphabet as their use is needed (predicate symbols, constants).

A term is defined in the same way as in first order logic. Atoms correspond to the atomic formulas, and a formula is defined in the same way as well. From here on, we shall introduce new properties and definitions.

Definition 2.29 (Default Literal)**.** *A default literal is an atom a, preceded by not, denoting default negation.*

Definition 2.30 (Literal)**.** *A literal is either an atom or a default literal.*

Definition 2.31 (Ground)**.** *An atom, term, literal, is ground if it contains no variables.*

Now that we have the basic definitions of logic programming, we are now able to define what a rule formally is.

Definition 2.32 (Rule)**.** *A rule is an ordered pair $\langle Head, Body\rangle$ of the type:*

$$p_0 \leftarrow p_1, ..., p_m, not\ p_{m+1}, ..., not\ p_n$$

such that $n \geq m \geq 0$, where Head, also denoted by $H(r)$ is the atom p_0, and the body, denoted by $B(r)$ is the set: $\{p_1, ..., p_m, not\ p_{m+1}, ..., not\ p_n\}$. We denote by $B^+(r)$ the set $\{p_1, ..., p_m\}$, the positive body of r, and by $B^-(r)$ the set $\{p_{m+1}, ..., p_n\}$, the negative body of r.

If the head is missing, the rule is called an integrity constraint. Also, if $m = n = 0$, the rule is called a fact.

With this, we can now define what a logic program is.

Definition 2.33 (Logic Program)**.** *A logic program is a set of rules. Also, a program is called definite if it has no default literals.*

2.3.2 Semantics

Before we can formally introduce the semantics, there are some important notions that need to be clarified.

The set of all ground terms of an alphabet $\mathcal{A}$ is called the *Herbrand Universe*. The set of all ground atoms that can be formed from predicate symbols and terms in its *Herbrand Universe* of an alphabet $\mathcal{A}$ is called the *Herbrand Base*. If we fix an alphabet $\mathcal{A}$, then we refer to the *Herbrand Base* of $\mathcal{A}$ by $\mathcal{H}$. Also, by grounded version of a logic program P, denoted $ground(P)$, we mean the (possibly infinite) set of grounded rules obtained from P, by substituting in all possible ways each of the variables in P, by elements of the *Herbrand Universe*.

We define the semantics of a logic program P by translating it into a closed first-order formula as follows:

Definition 2.34. *Let P be a logic program. The translation π with respect to a rule r and P is done as follows, where ω is the vector of the free variables of r:*

- $\pi(r) = \forall_\omega : (p_0 \subset p_1 \wedge ... \wedge p_m \wedge \neg\, p_{m+1} \wedge ... \wedge \neg\, p_n)$
- $\pi(P) = \bigwedge_{r \in P} \pi(r)$

A logic program P is satisfiable if and only if a model of $\pi(P)$ exists. Given a model I of P, $I \models P \Leftrightarrow I \models \pi(P)$, where $\models$ stands for satisfaction.

We now define what an *Herbrand Interpretation* of a language L is. Formally:

Definition 2.35 (Herbrand Interpretation)**.** *An interpretation of a logic program P is any subset of the Herbrand base $\mathcal{H}$ of P.*

A Herbrand Interpretation can also be seen as a First Order Interpretation, where all terms are interpreted by themselves.

Proposition 2.36. *Any interpretation I can equivalently been viewed as a function $I : \mathcal{H} \to V$, where $V = \{0, 1\}$, defined by:*

$$I(A) = \begin{cases} 0 & \textit{if} \quad \textit{not } A \in I \\ 1 & \textit{if} \quad A \in I \end{cases}$$

where A is an atom.

Definition 2.37 (Truth Valuation)**.** *If I is an interpretation, the truth valuation of $\hat{I}$ corresponding to I is a function $\hat{I} : C \to V$ where C is the set of all formulae of the language, recursively defined as follows:*

- *if A is a ground atom, then $\hat{I}(A) = I(A)$.*
- *if S is a formula then $\hat{I}(not\ S) = 1 - \hat{I}(S)$.*
- *if S and V are formulae then*
  - $\hat{I}((S, V)) = min(\hat{I}(S), \hat{I}(V))$.
  - $\hat{I}(V \leftarrow S) = 1$ *if* $\hat{I}(S) \leq \hat{I}(V)$, *and* 0 *otherwise.*

We now have enough information to formally define a Herbrand Model.

Definition 2.38 (Herbrand Model)**.** *A Herbrand Interpretation is called a Herbrand Model of a program P if for every ground instance of a program rule $H \leftarrow B$, we have $\hat{I}(H \leftarrow B) = 1$.*

Intuitively, a Herbrand Interpretation is called a Herbrand Model of a program P if all rules if P are satisfied in the Herbrand Interpretation.

From now on, we restrict ourselves to the *Herbrand Interpretation* and models, thus dropping the qualification of *Herbrand*.

Definition 2.39 (Classical Ordering)**.** *If $\mathcal{I}$ is a collection of interpretations, then an interpretation $I \in \mathcal{I}$ is called minimal in $\mathcal{I}$ if there is no interpretation $J \in \mathcal{I}$ such that $J \subseteq I$ and $J \neq I$. An interpretation I is called least in $\mathcal{I}$ if $I \subseteq J$ for all other interpretations $J \in \mathcal{I}$. A model M of a program P is called minimal (respectively least) if it is minimal (respectively least) among all models of P.*

Definition 2.40 (Minimal Model)**.** *An interpretation M is a minimal model of a program P if M is a model of P and is minimal among all other models of program P.*

Definition 2.41 (Least Model)**.** *An interpretation M is a least model of a program P if M is a model of P and is least among all other models of program P.*

Proposition 2.42. *Let P be a logic program. If P is a definite program, then it has a least model M.*

An atom A is true in program P, if and only if A belongs to its least model. Otherwise, A is considered false in P.

Consider the following program P:

$$ableMathematician(X) \leftarrow physicist(X).$$
$$physicist(einstein).\ \ president(cavaco).$$

One model of P could be: $\{president(cavaco), president(einstein), physicist(cavaco), physicist(einstein), ableMathematician(cavaco), ableMathematician(einstein)\}$. However, this is not the correct meaning of the program. Since we lack the information that *cavaco* is a physicist, we should considerate that he is not. Then, the least model of the program is: $\{president(cavaco), physicist(einstein), ableMathematician(einstein)\}$.

Stable models were first introduced in[GL88]. Informally, the idea is that when one assumes true some set of default literals, some consequences follow according to the semantics of definite programs. If the consequences completely corroborate the hypothesis made, then they form a stable model. Formally:

Definition 2.43 (Stable Model/Answer Set)**.** *Let I be a model of a logic program P. Let us denote by $\Gamma_P(I)$ the least model of the reduct of P with respect to I, denoted $\frac{\Pi}{I}$, defined as:*

$$\frac{\Pi}{I} = \{H(r) \leftarrow B^+(r) \mid r \in \Pi \wedge B^-(r) \cap I = \emptyset\},$$

where Π stands for $ground(P)$.

Then, a model I of a logic program P is a stable model of P if and only if $\Gamma_P(I) = I$.

We have now defined what a stable model is. However, there are also some properties that should be taken into consideration, and we present them next.

Definition 2.44 (Support)**.** *Given a model I for a ground program P, we say that a ground atom A is supported by P in I if there exists a rule r of $ground(P)$ such that $H(r) = A$ and $B(r) \subseteq I$.*

Proposition 2.45. *If I is an answer set of a program P, then all atoms in I are supported by P in I. [MS89, LRS97, BG94].*

3

Database Repair

Database repairing is a technique that allows consistency to be restored to a database whenever the database becomes inconsistent. If, for instance, a new integrity constraint is generated, leading the database to an inconsistent state, database repairing generates an update of the database restoring the consistency, by deleting faulty tuples, and/or adding new ones, creating a repaired instance, a database repair. In order for this technique to be possible, we are not just considering *inconsistency* but we are also considering that the database is incomplete.[1]

Throughout this chapter, we introduce the notion of inconsistency and formally describe the database repair problem, providing a few examples along as well.

3.1 Inconsistency

Inconsistency is a common phenomenon in the database world. There are many possible ways for a database to become inconsistent. One of them is due to the data being drawn from a variety of independent sources (as in data integration[Len02]). Another one, is due to the creation of new integrity constraints, with some information already stored in the database.

It is then important to formally introduce the concept of inconsistency, since it will be one of the main topics of this dissertation as well.

Definition 3.1 (Inconsistency)**.** *Given a database $D = \langle I, IC \rangle$, we say that D is consistent*

[1] *Incompleteness* here does not mean that the database contains *indefinite information* in the form of nulls or disjunctions[CS98]. Rather, it means that *Open World Assumption* is adopted, i.e., the facts missing from the database are not assumed to be false. Since the insertion of new tuples is possible, we must accept that the facts missing from the database are not false, or else, it would not be possible.

with respect to IC, if $D \models IC$, such that $D \models IC$ is true if and only if $\beta_t(I) \models$ IC; inconsistent otherwise[2].

Example 1. *Suppose we have the Customers and Accounts relations, belonging to a database D, represented in Table 3.1, and the inclusion dependency*

$$ic : \forall_{AId,CId} \exists_{Name} . [\neg Accounts(AId, CId) \vee Customers(CId, Name)]$$

Customers

CustomerId	Name
111	John
222	Peter
333	Anna

Accounts

AccountId	CustomerId
1	111
2	222
3	333
4	444
5	444
6	444

Table 3.1: Referential Integrity Problem

This database violates the integrity constraint defined, since $D \not\models ic$, since there is no tuple in relation Customers that has, as value of the attribute CustomerId, the value 444. Therefore, D is inconsistent with respect to ic.

Intuitively, a database is inconsistent if it does not satisfy the integrity constraints defined over it.

Now that we have defined what inconsistency is, let us formally introduce the notion of repair.

3.2 Repairs

To deal with inconsistency, database repairing proposes that a new database is created, replacing the older one, leading the database to a new consistent state. This new instance is called a *repair*. The new database generated will have the same name and schema of the original one. Only the content of the database is altered. Therefore, a *repair* is a new database, consistent with the integrity constraints defined over it.

Going back to *Example 1* , we saw that the database was in an inconsistent state. A possible repair, with respect to the original database, can be seen in Table 3.2.

As we can see, this new database is consistent with respect to the integrity constraints. In order to obtain this repair, we only took into consideration deletions. However, we may also use insertions. Consider the following database:

As we can see, the database presented in Table 3.3 is also a repair, since the integrity constraints are satisfied as well. However, we introduce the value $NULL$, which we

[2]Note that $\beta_t(I)$ is a subset of the Herbrand Base, so, it is a Herbrand Interpretation, and thus, also a First Order Logic Interpretation.

Customers

CustomerId	Name
111	John
222	Peter
333	Anna

Accounts

AccountId	CustomerId
1	111
2	222
3	333

Table 3.2: Referential Integrity Problem Repair 1

Customers

CustomerId	Name
111	John
222	Peter
333	Anna
444	NULL

Accounts

AccountId	CustomerId
1	111
2	222
3	333
4	444
5	444
6	444

Table 3.3: Referential Integrity Problem Repair 2

do not want to. Therefore, in order to allow insertions, an extra source of tuples (an additional database) must be explicitly introduced by the user. This new database shares the same schema of the original database, and does not have any integrity constraints defined. This exta database can be seen as a set of tuples that can be inserted in the original database in order to restore consistency. Going back to the previous example, consider that the user introduced the extra tuples depicted in Table 3.4.

Customers auxiliary

CustomerId	Name
444	Richard
555	Michael
666	Susan

Table 3.4: The Customers auxiliary relation

Then, according to this new source of information, consider the following database in Table 3.5. We now have generated a new repair, with respect to the original problem, by inserting tuples into the original database. This way, we avoid the use of $NULL$ values, although some additional effort is required, since extra tuples must be introduced in the process.

The next step would be to actually store this information in the database, replacing the previous instance with the new one, thus restoring consistency,

Let us formally define our database repair problem, and introduce the notion of a repair. Take into consideration that a source of extra tuples needs to be introduced by the user, in order to contemplate insertions.

Customers

CustomerId	Name
111	John
222	Peter
333	Anna
444	Richard

Accounts

AccountId	CustomerId
1	111
2	222
3	333
4	444
5	444
6	444

Table 3.5: Referential Integrity Problem Repair 3

Definition 3.2 (Database Repair Problem). *A database repair problem is a triple* $\langle D, E, IC_1 \rangle$ *where* D *and* E *are databases, such that* $D = \langle I_D, IC \rangle$ *is the main database and* $E = \langle I_E, \emptyset \rangle$ *is the source of new tuples, and* D *and* E *have the same database schema, and* IC_1 *is a set of integrity constraints, such that* $D \not\models IC \cup IC_1$.

The database repair problem consists on an existing database, where we want to insert new integrity constraints that leads the database to an inconsistent state. Therefore, since the database becomes inconsistent, a repair must be generated. We define a repair as follows:

Definition 3.3 (Repair). *Given a database repair problem* $P = \langle D, E, IC_1 \rangle$, *such that* $D = \langle I_D, IC \rangle$ *and* $E = \langle I_E, \emptyset \rangle$, *a repair, with respect to* P, *is a new database of the form* $D' = \langle I'_D, IC \cup IC_1 \rangle$ *such that:*

- $\beta_t(I'_D) \subseteq \beta_t(I_D) \cup \beta_t(I_E)$
- $D' \models IC \cup IC_1$

Although we have seen some possible repairs in Example 1, many more repairs may exist, besides the ones presented. Consider the following repairs, depicted in Tables 3.6 and 3.7. In the first one, we deleted all tuples from relation *CustomerId*. In the second one, and considering as well the source of extra tuples in Table 3.4, we added the tuple $\langle 444, Richard \rangle$, but also the tuple $\langle 555, Michael \rangle$ to the relation *Customers*.

Customers

CustomerId	Name
111	John
222	Peter
333	Anna

Accounts

AccountId	CustomerId

Table 3.6: Referential Integrity Problem Other Possible Repairs 1

Customers

CustomerId	Name
111	John
222	Peter
333	Anna
444	Richard
555	Michael

Accounts

AccountId	CustomerId
1	111
2	222
3	333
4	444
5	444
6	444

Table 3.7: Referential Integrity Problem Other Possible Repairs 2

This way, we can come up with some more repairs. However, the interesting ones are usually those that result from minimal change, i.e., where we change as little information as possible, which are called minimal repairs. In order to define what a minimal repair is, let us introduce some distance based measures, considering two distinct minimality criteria: minimality under set inclusion and minimality under cardinality of operations, where by operations, we mean deletions and/or insertions.

Definition 3.4 (Set Based Distance)**.** *Let D and D' be two databases over the same schema, such that $D = \langle I, IC\rangle$ and $D' = \langle I', IC\rangle$. The set based distance between D and D', denoted by $\Delta(D, D')$ is defined as:*

$$\Delta\left(D, D'\right) = (\beta_t(I) \setminus \beta_t(I')) \cup (\beta_t(I') \setminus \beta_t(I)).$$

Definition 3.5 (Set Based Closeness Relation)**.** *Let D, D' and D'' be three databases over the same schema. We say that D' is set closer to D than D'', denoted by $D' \prec^D_\Delta D''$, if $\Delta(D, D') \subset \Delta(D, D'')$.*

Definition 3.6 (Cardinality Based Distance)**.** *Let D and D' be two databases over the same schema, such that $D = \langle I, IC\rangle$ and $D' = \langle I', IC\rangle$. The cardinality based distance between D and D', denoted by $|\,(D, D')\,|$ is defined as*

$$|\left(D, D'\right)| = |\left(\beta_t(I) \setminus \beta_t(I')\right)| + |\left(\beta_t(I') \setminus \beta_t(I)\right)|$$

where $|R|$ denotes the cardinality of set R.

Definition 3.7 (Cardinality Based Closeness Relation)**.** *Let D, D' and D'' be three databases over the same schema. We say that D' is cardinality closer to D than D'', denoted by $D' \prec^D_{|.|} D''$, if $|\,(D, D')\,| < |\,(D, D'')\,|$.*

We are now able to formally define what a minimal repair is.

Definition 3.8 (Minimal Repair)**.** *Let $P = \langle D, E, IC_1\rangle$ be a database repair problem, and $\vartheta \in \{\Delta, |.|\}$ a closeness relation. A repair D' with respect to P is minimal with respect to ϑ if there is no repair D'' with respect to P such that $D'' \prec^D_\vartheta D'$*

Two different minimality criteria were taken into account in the previous definition. The minimality under set inclusion and the minimality under the cardinality of operations. In many of the cases, both of the minimality criteria will generate the same repairs. However, there are times when this is not true. To illustrate this, let us go back to the repairs previously shown. The repairs presented in Tables 3.2 and 3.5 are both minimal under set inclusion. However, the first one is not minimal under cardinality, whilst the second one is, since to obtain the first repair, we perform three operations, while we only need one operation to obtain the second repair. Also, the repairs presented in Tables 3.6 and 3.7 are not minimal under set inclusion nor cardinality.

Generating repairs is not a trivial task, specially if we consider the generation of minimal ones. The complexity of database repairing lies between the NP-hard and the Σ_2^p complexity classes[CM05]. Take into consideration that the study of the complexity of database repair is out of the scope of this dissertation.

Having formally introduced the database repair problem, it is important to know the origins of this problem, and what has been done to solve it. There has been a lot of research along the years, and there are several approaches to similar problems, which we shall discuss next, since they were of the most importance, in order to motivate our solution, the use of answer set programming, to address this problem.

4

Related Work

In order to deal with inconsistency in databases, two distinct techniques have been proposed: Consistent Query Answering and Database Repairing. In the first one, despite having an inconsistent database, the objective is to, when given a query to the database, return to the user only those tuples that are not in conflict with the defined integrity constraints. In the second one, the objective is to restore consistency to the database, by deleting and inserting tuples, or, in some cases, by updating values of attributes, if the database is inconsistent.

In this chapter, we shall discuss what has been done in these areas, with more emphasis on the Database Repair area, motivating the reader to the use of answer set programming to address the database repair problem.

We first discuss several approaches regarding Consistent Query Answering and, afterwards, Database Repairing.

4.1 Consistent Query Answering

Bry [Bry97], to the best of our knowledge, was the first author to consider the notion of consistent query answer in inconsistent databases. Informally, consistent answers are the result of a query posed to the database, that can be computed without using some data involved in an integrity constraint violation. However, Bry's approach is entirely proof-theoretic, and does not suggest any method for computing consistent query answers, not addressing as well the issues of the semantics associated with it, so it was not very clear what were the real implications of this technique.

In [ABC99], the authors addressed the problem of consistent query answering once more, formally introducing it, as well as introducing the notion of a repair (considering

only minimality under set inclusion). The authors defined consistent query answers as follows:

Definition 4.1 (Consistent Query Answer)**.** *A tuple $\bar{t}$ is a consistent answer to a query Q in a database instance D with respect to a set of integrity constraints IC, if $\bar{t}$ is an answer to the query Q in every repair D' of D with respect to the integrity constraints.*

Intuitively, a consistent query answer to an initial query contains those tuples that are true in every possible repair of the original database. The notion of repair was defined in the same way as we did, but only considering deletions as a repair primitive and minimality under set inclusion. Furthermore, in their work, the authors provide a logical characterization of consistent query answers in relational databases that may be inconsistent with the given integrity constraints. They state that a consistent answer to an initial query on an inconsistent database D, should be the same as the answers obtained, to the same query, from all possible repairs D' of D. In addition to this, they provided the first method for computing consistent answers, based on query modification. The authors also focused on the soundness and completeness of the proposed method. Their method works as follows: given a query Q, the method generates a modified query $T_\omega(Q)$, based on the notion of *residue* developed in the context of semantic query optimization, through an iterative algorithm, such that, the answers of $T_\omega(Q)$ are the same as the consistent answers of Q. However, this approach works only for quantifier free conjunctive queries, and only for a restricted set of integrity constraints, the universal constraints, where each variable in an atomic formula must be universally quantified. Inclusion dependencies were not possible to be considered using this approach. Based on this work, in [CB00], the authors, addressed once again the consistent query answering problem, designing an altered algorithm, $QUECA(.)$, for "QUEry for Consistent Answers", which, given a first order query Q, it generates a new query $QUECA(Q)$, whose answers are consistent with the integrity constraints, but, as opposed to the previous algorithm, it guarantees termination, soundness and completeness for a larger set of integrity constraints. However, this method was still not general enough, since they could only address a restricted class of integrity constraints (inclusion dependencies, for instance, were not addressed). The implementation of this algorithm was done using XSB[xsb].

The fact that the previous approach did not address inclusion dependencies showed that a new line of reasoning needed to be taken. And that is precisely what was done in [ABC00]. Here, with the topic of consistent query answer in mind, the authors proposed the representation of database repairs in form of a logic program with disjunction, under the stable models semantics[GL91, Prz91], such that, when given an initial query, an inconsistent database D and a set of integrity constraints IC, a repair program is built, such that there is a one-to-one correspondence between the stable models of the repair program and the repairs of D. Their approach consists in the direct specification of the database repairs in a logic programming formalism. They provide a mapping of the

database, along with the integrity constraints, into disjunctive logic programs with exceptions. The consistent query answers were given by performing the intersection of the answer sets generated, taking into consideration the initial given query. However, not all integrity constraints were covered in this approach, being the inclusion dependencies, again, left out. However, no practical results were shown. A deeper study was made in [ABC03] and [BB03], addressing the problem in more details, still using disjunctive logic programs to represent the database repair problem. Here, the authors considered inclusion dependencies, but still proposed further studies on that subject. Also, almost only binary constraints, i.e., constraints involving two relations, were studied. However, in all three of the previous works, the approach to CQA is based on the specification of all repairs, where each of them completely restores the consistency of the database, independently from the query that is posed and from the fact that it might have nothing to do with some of the violated integrity constraints. Also, disjunction is used in the logic programs, which, in the majority of the cases, increases the complexity of the problem. Furthermore, according to the mapping of the integrity constraints provided, an exponential number of rules may be created, increasing the size of the grounded program. The computation of all answer sets and the intersection of them was needed to consistently answer a query. It is that this is not desired, at all, since there may be a very large number of repairs to a specific problem, according to the size of the database and the integrity constraints defined, and the computation of all of them is not feasible in useful time. Therefore, the authors also suggested that it would be useful to specify and compute "repairs" that partially restored the consistency of the database, only with respect to the integrity constraints that are relevant to the query. Possibly, grounding techniques could be used in this case. Then, in order to reach such goal, other studies were made.

In [ABK00] and further on in [BBB01], the authors provide means to specify database repairs, again, using disjunctive logic programs, but also with annotation arguments, under the stable models semantics. These logic programs were based on a non classical logic, *Annotated Predicate Logic*. The objective was to embed both the database instance and the integrity constraints into a single theory where each predicate is replaced by a new predicate with an extra argument, the annotation argument. The role of the annotation argument is to enable the definition of atoms that can become true in the repairs or false, in order to satisfy the integrity constraints. By specifying how a database violates the integrity constraints, and how the database can become consistent, with respect to the integrity constraints, each atom can receive one of the constants depicted in Table 4.1. The authors show how to annotate integrity constraints (including inclusion dependencies), how to annotate queries and how to specify the repairs. The programs obtained by using *Annotated Predicate Logic* are simpler than the programs introduced in [ABC00, ABC03, BB03], in the sense that only one rule per integrity constraints is needed, whereas in the previous ones, may lead to an exponential number of rules. Therefore, the size of the ground program introduced here, is significantly reduced from the previous one. However, disjunction was still used in the specification of integrity constraints. The

Annotation	Atom	The tuple $P(\bar{c})$ is...
t_d	$P(\bar{c}, t_d)$	$P(\bar{c})$ is true in the database
t_a	$P(\bar{c}, t_a)$	$P(\bar{c})$ is advised to be made true
f_a	$P(\bar{c}, f_a)$	$P(\bar{c})$ is advised to be made false
t^*	$P(\bar{c}, t^*)$	$P(\bar{c})$ is true or is made true
t^{**}	$P(\bar{c}, t^{**})$	$P(\bar{c})$ is true in the *repair*

Table 4.1: Annotation Constants

way to obtain consistent query answers was to run the repair program along with a query program under the skeptical stable models semantics, that sanctions as true what is true in all models.

Since the way to obtain consistent query answers still lied in the intersection of the stable models, implementation issues, once more, arose, due to the high complexity of the whole process. Then, the authors discussed some optimizations of repair programs, examining certain program transformations that can lead to programs with lower computational complexity. Also, the authors discussed optimizations that avoided the computation of repairs when a query is to be answered. In fact, since the current approach relied in finding the ground atoms that belonged to all stable models of a repair program, the authors suggest that the problem of developing query evaluation mechanisms for disjunctive logic programs that are guided by the query, most likely containing free variables and then expecting a set of answers, like magic sets deserved more attention from the logic programming and database communities, which led to the following studies.

Work done in [MB05, MB07] explored the usage of magic sets in the computation of the consistent query answers, still with the annotation semantics, as describe in [ABK00, BBB01]. Here, the magic set techniques for logic programs with stable model semantics take as input a logic program - in this case the repair program - and a query expressed as a logic program that has to be evaluated against the repair program. The output is a new logic program, the magic program, with its own stable models, that can be used to answer the original query more efficiently. By using such techniques, there are less (and smaller) generated stable models. A skeptical reasoning is still needed to answer the queries, however, the process is now much more efficient. The authors extended magic set techniques, in order to deal with disjunctions. All this work culminated in a PhD thesis[Can07], where this approach was fully studied along with further optimizations to the whole process. Notice that the purpose of these works is not the computation of database repairing, but to provide consistent answers regarding a given query. In [MB05, MB07, Can07], there was no need to compute all the repairs and then to perform the intersection of all repairs. Those approaches introduced a more efficient way to reach such goal.

So far, the database repair programs only considered, as repair primitives, deletions and/or insertions. Nevertheless, there have also been some studies where updates were

considered. The motivation of this kind of repair primitive is that, whenever a tuple is violating an integrity constraint, it may contain information that is still correct and does not necessarily need to be deleted (or must not be deleted at all). That way, by updating the faulty values (substituting one constant by another constant/variable), we can maintain the correct information. In [Wij03], tuple updates were inserted as a repair primitive, and the notion of consistent query answering was adapted to this new method. In [Wij05], this approach was studied more deeply, providing some complexity results of this new approach. However, these updates considered the substitution of constants by variables, which introduced ambiguity in the database, despite restoring consistency. Also, values of attributes could by updated by the $NULL$ value, which we, in our approach, do not want to be present in the domain of any attribute.

As we could see, consistent query answering and database repair are usually two distinct problems that are, in a way, related. Usually, in order to perform CQA, a repair program is generated (when dealing with answer set programming). However, specialized algorithms have been developed in order to not have to compute the repairs to obtain the consistent query answers. Also, in all of these works, database repairs have been defined under a minimality criteria - minimality under set inclusion. However, in practical implementations, it was not shown how to address this problem, which, in our approach, we want to address. Furthermore, only minimality under set inclusion was explored. It may not be the best minimality criteria to use in a real database environment.

Although answer set programming was used in some of the previous approaches, disjunction was used in the logic programs created. Also, not all integrity constraints were discussed in the mapping of the consistent query answering problem. Furthermore, we think that a more effective mapping can be done. We also find that minimality under cardinality of operations should be discussed in greater detail.

Since our main objective is the database repairing technique, let us drive away from this path, and focus on this technique instead.

4.2 Database Repair

Unfortunately, there has been a lot more work and study in the consistent query answering part rather then in the database repair itself, thus more results are presented in this area. However, there have been some studies in the the later as well, and we shall go over them. Also, some applications have been developed to deal with real life database repair problems, whose results we shall discuss as well.

A very interesting approach on database repair was proposed in [GGZ01] and further developed in [GGZ03], where a framework for repairing databases was analysed in great detail. There, the authors provide examples of mappings of the database repair problem into logic programs (under the stable model semantics), focusing on the mapping of the integrity constraints. Furthermore, the authors present a very interesting particularity, the repair constraints property, which is the possibility to restrain the number of repairs.

They give preference to certain data with respect to others and define which repairs are feasible. It gives the possibility to restrain deletions or insertions in a specific relation, together with some additional properties that need to be verified. They also introduce the notion of prioritized updates, which are rules which give the possibility of expressing preferences among deletions and insertions, and consequently, among repairs. For instance, consider a relation that keeps the information about the employees of an enterprise. If a tuple of that relation has to be deleted due to a violation of an integrity constraint, we may prefer to delete the tuple of an employer with higher salary. However, such prioritized rules increase the complexity of the repair process. The authors also show that their approach is sound and complete.

The computation of the repairs is carried out by rewriting:

- integrity constraints into disjunctive rules,
- repair constraints into logic rules,
- prioritized update rule into logic rules.

In this dissertation, we did not want to address the problem using logic programs with disjunction, which, generally, increases the complexity of the problem we wish to address. Also, in their mapping of the database repair problem into a logic program, the authors only considered universal integrity constraints. Although key constraints, functional dependencies and denial constraints can be seen as universal constraints, there could be a better mapping of the integrity constraints, such that the computational time needed to reach the repairs is reduced.

Nevertheless, this work allowed to comprehend a little better the use of logic programs to aid the repair process, and introduced some very interesting particularities, such as the repair constraints and the prioritized update rules. However, it was just a theoretical study. No practical implementation of the framework was done. Also, it is not said how to incorporate minimality under set inclusion in their work. Moreover, they do not take into account that, for instance, key constraints, functional dependencies are distinct classes of integrity constraints, since they can both be expressed through a universally quantified constraint. If considered separately, a more efficient mapping can be done. Also, and once more, disjunctions are used in the logic programs.

In [FPL+01], the authors show a real application of the database repair problem. To repair the database, they only considered updates, where the attributes of a relation had a finite domain, whose cardinality was relatively small. The authors suggested the application of database repair in census data, which are in agreement with the recommendations for the 2000 European censuses of population [fEotEC98]. These censuses data provide valuable insights on the economic, social and demographic conditions and trends occurring in a country. The collected information provides a statistical portrait of the country and its people.

The idea behind the census is that each family completes a questionnaire, which includes the details of the head of the household (the householder) together with the other people living in house (being them family related or not). However, when collecting the questionnaires, some of them may be incomplete of have inconsistent information. Imagine a married person who claims to be only 6 years old. Obviously, this is inconsistent. It is easy to accept that the deletion of tuples is not wanted, since it would totally ruin the statistical purpose of the census. Therefore, the authors propose a framework to correct this errors by updating the values, by introducing a way to encode the problem into a disjunctive logic programming language. The authors provide as well a small running example while mapping the problem into a logic program.

In their work, a questionnaire is $\mathcal{Q}$ is defined as a pair $\mathcal{Q} = \langle r, E \rangle$, where r is a relation and E is a set of integrity constraints that must always be satisfied in r. An example of an integrity constraint may be: any married person should be at least 16 years old. However, this is not enough to perform updates, since there may exist some possible updates to a repair that do not make sense, for instance: in the previous case of the married person who is 6 years old, a possible repair would be to change the person's age to 150, which clearly does not make sense. The authors were only interested in the minimal repairs that did not alter the statistical properties of the census. Therefore, they also introduced the concept of preferred rules, which can be encoded through a set of first order formulas, used for expressing some preferences over a repaired relation. An example of a preferred rule may be: it is likely for a married person living in the household, whose relationship with the householder is unknown, to be his/her spouse. This way, preference rules are used to establish an ordering within the minimal repairs of a questionnaire. Intuitively, when some value is changed in order to repair inconsistencies and/or to replace $NULL$ values, the repair should satisfy as many preferences rules as possible.

The encoding is shown not in a formal way, but accompanied by a small running example. However, the authors describe how to address the minimality. The authors performed several tests using the *DLV* solver, and that the results of such experiments showed the feasibility of this approach with realistic data.

Once more, this work is based on logic programs with disjunctions, which we do not wish to incorporate in our approach. Also, despite being a real practical application, this application may mislead the users into believing that updates are the best way to perform database repairs. The use of updates as a repair primitive, may introduce ambiguity in the repairs, since many studies allow the insertion of variables or the $NULL$ value. In this case, updates are only possible since the domains of the attributes are relatively small, being then possible to create rules to express how to repair a specific inconsistency. For instance, if a person does not specify the gender, a rule can be expressed saying that the value of that attribute is either *male* or *female*. However, if the domain of an attribute is very big, such approach is not feasible. Notice as well, that, the authors do not consider the most used integrity constraints in a database management system. They need to introduce specific rules to every situation that they wish to cover. In this dissertation,

despite wanting to allow the user to freely specify new integrity constraints, we want to focus in the integrity constraints provided directly by the database management system as well.

The last approaches were all based on the representation of the database repair problem into logic programs. However, there have also been other approaches to database repair without the use of logic programs, or together with some other methodologies. We shall now go over some literature in database repair that relies on other technologies to address this problem.

In [GL97], the authors propose one of the first methods for database repairing. This was one of the initial works done in this area. Nevertheless, a very interesting approach, based on model-diagnosis was done in this work. Here, the authors do not consider only existing components in the database (positive facts), but also missing components (negative facts), that are necessary to be inserted in order to satisfy, for example, inclusion dependencies. Their approach allows the use of deletions and insertions, as repair primitives. Then, the objective of their approach is:

- To determine reasons for the constraint violations in an inconsistent database by computing the minimal sets of positive and negative facts that account for all violations in the database;
- To characterize schemas for possible repair actions that can be associated with such facts.

In order to accomplish the previous, the authors present a sound and complete algorithm for enumerating possible minimal repair transactions for an inconsistent database. It performs an iteration of diagnosis and repair of a constraint violation in a breadth-first search manner, through hypothetical databases (which represent hypothetical repaired databases with respect to a particular integrity constraint).

Being one of the first approach to database repairing, there are several issues regarding this approach. First, the authors consider a small fraction of integrity constraints. They only consider integrity constraints that are defined by means of base predicates. Also, denial constraints cannot be introduced in his approach, and these kind of constraints can be very powerful, so it would be interesting to allow such integrity constraints. Furthermore, if more than one dependency exists between two integrity constraints, the authors force the user to weight all possible ways to repair the database and indicate which one he wishes to use. If another way to repair was chosen, the final repair may be different. Therefore, this approach does not present all possible way to repair the database to the user. An automatic way to compute all minimal repairs would be interesting to develop.

Although the authors developed a real implementation of the algorithm, no results have been presented. So, despite being a sound and complete algorithm, we cannot infer about the scalability of such an approach. However, some particular aspects of this approach are very useful and important, which are:

- Determination of facts as well as missing facts that contribute to the different constraint violations;

- Possibility for the user to choose a repair strategy, following a repair goal (repairs minimal under set inclusion or repairs minimal under cardinality on the number of operations).

More recently, in [KL09], the authors suggest an algorithm that approximates optimum repairs for functional dependencies violations. The proposed algorithm only takes into account functional dependencies. The repairs are based on a minimality criteria, concerning the number of operations performed. However, only updates of values of attributes are allowed, i.e., the values of the attributes may be updated with another constant, or with a variable, which comes from an infinite set of variables. The algorithm relies on graph theory, by building a conflict hypergraph of an inconsistent database instance that represent the conflicts of the attributes of all tuples from a relations. Then, a minimum vertex cover is applied and the algorithm solves the inconsistencies that may exist.

This approach is merely a theoretical study. No practical implementation has been done. Furthermore, it introduces an algorithm to perform database repairing taking into consideration only functional dependencies. By only considering this kind of integrity constraint, it is very limited. Although the algorithm provided, if implemented, may have great performance, it lacks generality. We must take into consideration that a DBMS implements more integrity constraints than just functional dependencies. Also, in this algorithm, updates have been allowed as the only repair primitive. Because of that, the algorithm can update a constant by a variable, whenever the algorithm cannot suggest a value for that constant. Then, unknown values may be introduced in the database. More importantly, if we have a functional dependency of the form $X \rightarrow Y$, where X is compose by one attribute and Y by a set of attributes, for example, the algorithm may update the value of X to a variable, which is not desired at any time, since we lost notion of what the integrity constraints is actually restricting in reality.

In [BIG10], and following the notion of repair and some lines of reasoning introduced in [KL09], the authors propose a different repairing approach, still based on value updated, rather that tuples deletions, and again, only considering functional dependencies. The algorithm introduced by the authors, roughly speaking, "randomly" generates new possible repairs. The algorithm relies on the following: for any two tuples t_1, t_2 that violate a functional dependency $X \rightarrow Y$, it is enough to modify $t_1[Y]$ so it equals $t_2[Y]$ (or vice-versa), or modify an attribute $B \in X$ in either t_1 or t_2, so that $t_1[B] \neq t_2[B]$. Generalizing this observation, if a set of values of a set of attributes $\mathcal{C}$ does not violate any functional dependencies, consistency of the set $\mathcal{C} \cup C$, for any value C of an attribute, can always be enforced by modifying C.

The objective behind the algorithm is to continuously select "random" values from a database and keep storing those values. If, at any point, a value is selected that, together

with the already stored values, violates the functional dependency, that value is updated by another constant or by a variable. At the end, minimal repair, under cardinality on the updates performed, are generated.

In this approach, we have, once more, the problem of generality versus performance. Also, although the authors show that the algorithm can generate all minimal repairs, it takes several iterations of the algorithm to reach all minimal repairs. Moreover, the user may not know that all possible repairs have been generated, since the algorithm is based on a "random" choice of values of attributes. Furthermore, this algorithm allows the update of a value by a variable. Once more, this variable means that an unknown value is there, which may not be desired in a database. In their approach, the authors allow the users to specify hard constraints, i.e., to specify that some values of attributes cannot be updated during the repair process, which is a very interesting feature.

In [CM11], the authors came up with a new idea, with respect to the database repair problem. Instead of simply repairing faulty tuples, the authors suggest the correction of integrity constraints (in their approach, only functional dependencies were taken into account). They defend that in modern applications, constraints may evolve over time, as application or business rules change, as data is integrated with new data sources, or as the underlying semantics of the data evolves. In this work, the authors developed an algorithm to capture the inconsistencies and repair them. The main objective of their approach is: given a database instance I that is inconsistent with respect to a given a set of functional dependencies Σ, they want to find a set of low cost data repairs and a repair of Σ (the repair of integrity constraints), such that the repaired database is consistent with the repaired set of integrity constraints.

Since only functional dependencies are considered, all integrity constraint are of the form $X \rightarrow Y$. If there is a violation of an integrity constraint, it means that there exist two distinct tuples t_1 and t_2, such that $t_1[X] = t_2[X]$ and $t_1[Y] \neq t_2[Y]$. One possible repair is simply to change the values of $t_2[Y]$ to $t_1[Y]$. Another possible repair, is to change the values of $t_1[X]$, to the values of another tuple $t_3[X]$, where $t_3[Y] = t_2[Y]$. However, when dealing with these kinds of updates (the left side of the implication), the authors only consider values for X that are supported by other tuples. This way, ambiguity is not introduced in the database. Only value updates are considering when repairing the data.

In order to perform a repair of an integrity constraint $X \rightarrow Y$, the users propose the addition of another attribute A to X, such that $X \cup A \rightarrow Y$.

In this algorithm, optimality takes into consideration both tuples repairs or constraints repairs, so that the optimum repair is the one that performs less changes. Also, the algorithm provided is a greedy algorithm, i.e., an algorithm that relies on the use of heuristics.

There are several aspects that need to be pointed out in this study. Once again, only functional dependencies are being taken in consideration. Therefore, the authors traded performance with generality. When updating the values of the attributes of the right side of an inclusion dependency, the authors simply update the values of a set of attributes by the exact same values of another set of attributes. This may not be desired in

many cases. Consider we have the relation depicted in Table 4.2, where we have a relation to store information about people and the inclusion dependency $Id \rightarrow Name,\ Age$. By the algorithm provided, the first tuple would become $\langle 1, Anna, 22\rangle$ or the second tu-

Id	*Name*	*Age*
1	Michael	27
1	Anna	22

Table 4.2: Inconsistent Person Relation

ple would become $\langle 1, Michael, 27\rangle$. Either way, we would be losing information. If, on the other hand, the algorithm suggested that the functional dependency should become: $Id, Name \rightarrow Age$. Then, we may ask why are the integrity constraints necessary? Integrity constraints are supposed to be permanent, in order to have consistent information at all times. If we are assuming that the integrity constraints may change in time, then why should we even use them?

Lastly, since the algorithm describe is a greedy algorithm, heuristics are used. By having heuristics, the tuning of some parameters is extremely important, in order to have accurate answers, otherwise, the algorithm can be very fast, and produce weak repairs, or, the repairs may be extremely efficient, but the time needed to reach such solutions is very high. There is always this kind of trade-off between quality and performance. Since it is a greedy algorithm, it may fail to compute the best optimum repair, simply because it does not exhaustively look for repairs. This way, completeness is not achieved.

In [SMG10], the authors proposed the use of the *Argumentation* techniques to provide a better understanding of the reasoning process behind database reparation. In their approach, they provide means to identify, represent and resolve the conflicts between tuples in an inconsistent database. The objective goes through building an argumentation tree, where arguments attack each other. In the end, each branch corresponds to a possible way of restoring consistency to the database system. They also provided a mechanism to ensure optimal repair checking.

Being this work purely theoretical, the authors introduced a new line of research that embraces the database repair and argumentation research areas. But, to this point, we cannot yet infer about the use of argumentation in this kind of technique, since we lack enough results to do it, and this work is still just a preliminary study.

Work on the area of data cleaning has also been done, and some studies present very interesting results. The purpose of data cleaning it to detect and remove errors and inconsistencies from data in order to improve the quality of the data. Data quality problems are present in databases, due to misspellings during data entry, missing information or other invalid data [RD00]. This is what is called *dirty data*. The way to correct this *dirty data* may be, as in our approach, by rejecting tuples (deleting them from the database), or correcting them (updating values of attributes). However, the process is computationally expensive on very large databases. Besides considering inconsistency with respect to the

integrity constraints, data cleaning considers as well other sources of errors. Therefore, some work done in data cleaning is very interesting and strongly related to database repairing. There are several popular methods used in data cleaning, such as:

- **Parsing:** detection of syntax errors;
- **Data Transformation:** mapping of the data from their given format into the format expected by the appropriate application (value conversions for instance);
- **Statistical Methods:** by analysing the data using values of mean, standard deviation, range or clustering algorithms, it is possible to find values that are unexpected and thus erroneous;
- **Duplicate Elimination:** determines whether data contains duplicate representations of the same entity.

The latter is a very interesting problem, and is strongly related to some database repair problems. We now present some work done in this area.

In [LLL00, LLL01], the authors introduced *IntelliClean*, a knowledge-based intelligent data cleaner, which is an application built do deal with duplicated tuples detection. They developed a framework that provides a complete strategy for duplicate elimination in dirty databases with duplicate (or seemingly duplicate) tuples. Their framework consists on a three stage algorithm:

- Pre-Processing Stage - tuples are first conditioned and scrubbed of any anomalies that can be detected at this stage. For instance, inconsistent abbreviations used in the data can be resolved at this stage (e.g. occurrences of '1' and 'A' in the sex attribute will be replace by 'Male', and occurrences of '2' and 'B' in the sex attribute will be replace by 'Female'). The output of this stage will be a set of conditioned tuples which will be input to the processing stage.
- Processing Stage - the conditioned tuples are next fed into an expert system engine together with a set of rules. These rules will fall into one of the following categories:
  - Duplicate Identification Rules - these rules specify the conditions and criteria for two tuples to be classified as duplicates. Tuples are then classified as duplicates with a *Certain Factor*;
  - Merge/Purge Rules - these rules specify how the duplicates tuples are meant to be handled;
  - Update Rules - these rules specify the way data is to be updated in a particular situation;
  - Alert Rules - the user might want an alert to be raised when certain events occur. Such rules may be useful when the DBMS in which the data resides does not support the checking of constraints.

- Human Verification and Validation Stage - human intervention is required to manipulate the duplicate tuples groups for which merge/purge rules are not defined.

The algorithm used in this framework is the Rete Algorithm [For82]. It is an efficient method for comparing a large collection of patterns to a large collection of objects.

The rules, which form the knowledge-base of the framework, are written in Java Expert System Shell language.

This framework, however, may introduce false-positives, i.e., may consider that two tuples are considered duplicate when, in reality, they are not. Therefore, the authors introduced some threshold in order to reduce the number of wrongly merged duplicated groups. Also, according to the rules generated, the application can be more accurate or less accurate, but there is a trade-off. Being more accurate, means that more precise rules need to be defined, increasing the computational time. Relaxing the rules, would decrease the computational time, but increase the amount of false-positives as well.

This framework, as most of the studies already presented, only relies on specific integrity constraints. Here, the authors only studied the case of duplicate tuples, which, are only a bigger problem when dealing with database merging. Once more, it is built around a very particular and specific problem. Generality of the algorithm is not achieved. Also, it requires human intervention in some cases that rules were not defined.

In [BSIBD09, BSI+10], the authors introduce *ProbClean*, a probabilistic duplicate detection system. It is a system that treats duplicate detection procedures as data processing tasks with uncertain outcomes. They concentrate on a family of duplicate detection algorithms that are based on parametrized clustering. The motivation behind their idea is that, generating a single repair necessitates resolving the uncertain cases deterministically. Once the repair is chosen and updated, information is lost. Sometimes, it may be the case that we do not wish this to happen. Therefore, the authors provide a system that keeps track of multiple repairs, by probabilistically modelling possible repairs, allowing uncertainty when deciding the duplicate tuples. The challenges involved in the process are:

- Generation of Possible Repairs
- Succinct Representation of Possible Repairs
- Query Processing

For this purpose, the authors defined the notion of Duplication Repairs as such:

Definition 4.2 (Duplication Repair)**.** *Given an unclean relation R (a relation that contains duplicate tuples), a repair X is a set of disjoint tuple clusters,$\{C_1, ..., C_m\}$ such that $\bigcup_{i=1}^{m} C_i = R$.*

A repair X partitions R into disjoint sets of tuples that cover R. By coalescing each set of tuple in X into a representative tuple, a clean (duplicate free) instance of R is obtained.

In their approach, *ProbClean* generates all possible repairs corresponding to a set of possible parameter settings of any fixed parametrized clustering algorithm. To model the possible repairs, *ProbClean* creates a new relation, R^c, which are sets of $c-tuples$, where each $c-tuple$ is a representative tuple for a cluster of tuples. Attributes of R^c have all the attributes of R, plus two special attributes: C and P. Attribute C of a $c-tuple$ is the set of tuples identifiers in R that are clustered together to form the $c-tuple$. The attribute P of a $c-tuple$ represents the parameters settings of the clustering algorithm used to generate the cluster represented by the $c-tuple$. The authors present some clustering algorithms and provide some examples on how to build the space of repairs. Also, when merging two or more distinct tuples into one representative tuples, there are also some measures needed to merge the values of the attributes. The authors also provide some means to do this operation.

ID	...	Income	C	P
CP1	...	31k	{P1,P2}	[1,3]
CP2	...	40k	{P3,P4}	[0,10]
CP3		55k	{P5}	[0,3]
CP4	...	30k	{P6}	[0,10]
CP5	...	39k	{P1,P2,P5}	[3,10]
CP6	...	30k	{P1}	[0,1]
CP7	...	32k	{P2}	[0,1]

A(*Person*, τ : *U*[0,10])

ID	Name	ZIP	Birth Date	Income
P1	Green	51359	781310	30k
P2	Green	51358	781210	32k
P3	Peter	30128	870932	40k
P4	Peter	30128	870932	40k
P5	Gree	51359	19771210	55k
P6	Chuck	51359	19460924	30k

Figure 4.1: *ProbClean* repair

Figure 4.1 shows an example of a clean relation, that captures possible repairs generated by a clustering algorithm $\mathcal{A}$, with a uniformly distribute parameter τ in the range $[0, 10]$. For example, if $\tau \in [1, 3]$, the resulting repair of Relation *Person* is equal to $\{\{P1, P2\}, \{P3, P4\}, \{P5\}\{P6\}\}$. The cluster $\{P1, P2, P5\}$ is generated if the value of parameter τ belongs to the range $[3, 10]$.

The authors in this work also provide means to query the repairs of the unclean relation. They define the selection, projection and join operators over repaired clean relations.

Since the ultimate objective is to obtain one good clean relation, the authors provide means to compute the most probable relation, via a probability function define over $c-tuples$. This function determines the membership probability of a $c-tuple$ given its parameter settings.

Although, at the beginning, it may seem a interesting approach, what would happen if the parameters used in the clustering algorithm generated a high number of repairs? It would no be efficient to maintain them all and reason over them. This brings us to the problem of developing algorithms that rely on the parametrization of some values. An inexperienced user may choose very bad parameters, resulting in very bad repairs as well. Once more, there is also the problem of the lack of generality in the algorithm, since

if focus in a very specific problem: duplicate tuples detection.

As we saw throughout this chapter, there are several approaches to deal with the database repair problem. However, in every one of them, we pointed some particularities that were not considered, or were considered but we thought that they were not approached in the best way. Some approaches only considered a small fraction of integrity constraints. Others considered that integrity constraints were subject to changes. Others considered the use of greedy algorithms in order to obtain repairs, not providing a sound and complete approach. For those approaches that relied on the use of answer set programming, disjunction was used, and a substantial number of integrity constraints were replaced by universal constraints, where we think that it would be best if the others integrity constraints were considered separately. Furthermore, most of the approaches defined a repair using a minimality criteria but did not provide any practical implementation considering that, nor gave approach on how to address such problem. Also, most of the studies did not consider minimality under cardinality of operations. For this purpose, in this dissertation, we wish to address the creation of new integrity constraints (all that can be expressed in answer set programming) that may lead the database to an inconsistent state, returning correct and minimal (under set inclusion or under cardinality of operations) repairs to the user. As we have also seen, methods that relied on the use of answer set programming allowed to represent the database repair problem in a very adequate manner, being also sound and complete methods. Also, as far as we know, there isn't, yet, any real application to compute the repairs and actually restore consistency to the database. We need to specifically know which tuples are meant to be deleted and which tuples are meant to be inserted. With that information, we can generate the adequate SQL code and actually perform those operations into the database, thus restoring consistency. For this purpose, we shall use the declarativeness and expressiveness of answer set programming, since, as we saw throughout this chapter, it is a suitable approach to the problem.

5

Database Repair with Answer Set Programming

In this chapter, we present our solution to the database repair problem. We first present the way to map a repair problem into a logic program, showing as well the soundness and completeness of it, i.e., we prove the all solutions are repairs, and we prove that all repairs are generated. Afterwards, we present the mapping of a minimal repair problem, where minimality issues are considered.

Throughout this chapter, we shall adopt some new notation as well. We will denote by $\bar{x}, \bar{y}, \bar{z}$, sequences of pairwise distinct variables with appropriate arity, of the form $\bar{x} = \langle x_0, x_1, ..., x_n \rangle$ such that x_i denotes the i^{th} component of $\bar{x}$ and where x_0, y_0,z_0 represent the special extra attribute, the $Rowid$ of the respective tuple. Also, we will denote by $\bar{c}, \bar{d}$, sequences of pairwise distinct constants with appropriate arity, of the form $\bar{c} = \langle c_0, c_1, ..., c_n \rangle$, such that c_i denotes the i^{th} component of $\bar{c}$ and where c_0, d_0 represent the value of the special extra attribute, the $Rowid$ of the respective tuple. Also, in the answer set part, the symbol '$\wedge$' stands for the symbol ',' as a matter of simplicity in the transformation function.

5.1 General Approach

In this section, we show the general approach to our repair problem. We provide some fundamental definitions that will be used from here on. We begin by introducing the mapping of a repair problem into a logic program. We also introduce some definitions in order to help us prove the soundness and completeness of our approach.

In the sequel, we transform a database repair problem into a logic program, with

respect to an initial database $D = \langle I_D, IC \rangle$, more specifically, into an answer set program. The alphabet of the language is the one introduced in Section 2.3, and, following our mapping procedure, the following predicate symbols should always be present in the alphabet as well:

- Predicate symbols:

$$\begin{aligned} \{p_N, p\_keep_N, p\_keep_N^i \mid N \in \mathcal{N}\} \cup \{insert, delete, n\_insert, n\_delete\} \cup \\ \{d_a^N \mid N \in \mathcal{N} \wedge a \in \mathcal{ATT}\} \cup \\ \{aux_{N_1,N_2}^{\langle \kappa_1,...,\kappa_k \rangle, \langle \varepsilon_1,...,\varepsilon_k \rangle} \mid N_1, N_2 \in \mathcal{N} \wedge k \in \mathbb{N} \wedge \kappa_i, \varepsilon_i \in \mathcal{ATT}\} \end{aligned}$$

Also, recall that $\mathcal{ATT}$ is the set of all names for the attributes of all relations in a database, and $\mathcal{N}$ is the set of all names of relations in a database.

We wish to transform a database repair problem into a logic program, such that there is a one to one correspondence with the generated models and the database repairs. The core of this transformation is the mapping of the integrity constraints into the logic program. It is important to notice that there is a strong relation with the formal first order definitions of the integrity constraints, described in Chapter 2.2. We must also take into account the possible insertion of new tuples to deal with inconsistency.

Definition 5.1 (Problem Transformation Function)**.** *Let $P = \langle D, E, IC_1 \rangle$, be a database repair problem, such that $D = \langle I_D, IC \rangle$ and $E = \langle I_E, \emptyset \rangle$. Let $\varphi(P)$ be the logic program created by performing the following operations:*

1. *Facts:*

   - *For every atom $N(t_1, t_2, ..., t_n)$ in $\beta_t(I_D)$, we add the following, as a fact, to $\varphi(P)$:*

   $$p_N(t_1, t_2, ..., t_n). \tag{5.1}$$

   - *For every atom $N(t_1, t_2, ..., t_n)$ in $\beta_t(I_E)$, we add the following, as a fact, to $\varphi(P)$:*

   $$p\_keep_N^i(t_1, t_2, ..., t_n). \tag{5.2}$$

2. *Repair Generator:*

   - *For every relation $N \in \{N' \mid \langle N', C, S \rangle \in I_D\}$, we add the following rules to the program:*

   $$delete(x_0) \leftarrow not\ n\_delete(x_0) \wedge p_N(\bar{x}). \tag{5.3a}$$

   $$n\_delete(x_0) \leftarrow not\ delete(x_0) \wedge p_N(\bar{x}). \tag{5.3b}$$

   $$p\_keep_N(\bar{x}) \leftarrow p_N(\bar{x}) \wedge not\ delete(x_0). \tag{5.3c}$$

- *For every relation* $N \in \{N' \mid \langle N', C, S\rangle \in I_E\}$*, we add the following rules to the program:*

$$insert(x_0) \leftarrow not\ n\_insert(x_0) \wedge p\_keep^i_N(\bar{x}) \tag{5.4a}$$

$$n\_insert(x_0) \leftarrow not\ insert(x_0) \wedge p\_keep^i_N(\bar{x}) \tag{5.4b}$$

$$p\_keep_N(\bar{x}) \leftarrow insert(x_0) \wedge p\_keep^i_N(\bar{x}). \tag{5.4c}$$

3. *Functional Dependencies/Keys: For every functional dependency* $FD(N, A, B) \in IC \cup IC_1$*, we add the following rules to the repair program:*

   For each $j \in \#^N_B$ *we add a rule of the type:*

$$\bot \leftarrow p\_keep_N(\bar{x}) \wedge p\_keep_N(\bar{y}) \wedge \bigwedge_{i \in \#^N_A} x_i = y_i \wedge x_j \neq y_j. \qquad (FD^M(N, A, B))$$

4. *Inclusion Dependencies: For every inclusion dependency* $IND(N_1, N_2, A, B) \in IC \cup IC_1$*, we add the following rules to the repair program:*

$$aux^{A,B}_{N_1,N_2}(\bar{x}) \leftarrow p\_keep_{N_1}(\bar{x}) \wedge p\_keep_{N_2}(\bar{y}) \wedge \bigwedge_{i=1}^{k} x_{\#^{N_1}_{\kappa_i}} = y_{\#^{N_2}_{\varepsilon_i}}. \qquad (IND^M(N_1, N_2, A, B)a)$$

$$\bot \leftarrow p\_keep_{N_1}(\bar{x}) \wedge not\ aux^{A,B}_{N_1,N_2}(\bar{x}). \qquad (IND^M(N_1, N_2, A, B)b)$$

5. *Check Constraints: For every check constraint* $CC(N, \kappa, \theta, V) \in IC \cup IC_1$*, we add the following rules to the repair program:*

$$\bot \leftarrow p\_keep_N(\bar{x}) \wedge x_{\#^N_\kappa}\ \theta\ V. \qquad (CC^M(N, \kappa, \theta, V))$$

6. *Domain Constraints: For every domain constraint* $DoC(N, \kappa, Do) \in IC \cup IC_1$*, where* $Do = \{val_1, val_2, ..., val_n\}$*, we add the following rules to the repair program:*

   For every $val_i \in Do$ *we add a rule of the type:*

$$d^\kappa_N(val_i). \qquad (DoC^M(N, \kappa, Do)a)$$

   We then add the integrity constraint rule:

$$\bot \leftarrow p\_keep_N(\bar{x}) \wedge not\ d^\kappa_N(x_{\#^N_\kappa}). \qquad (DoC^M(N, \kappa, D)b)$$

In the first step of the previous definition, we simply import all the tuples of the database and add them as facts to the logic program. We do the same method with the tuples from the extra source of tuples.

In the second step, in order to determine the new database instance, we consider that all tuples from D are possible tuples to be deleted and that all tuples from E are possible tuples to be inserted. We are then generating all combinations of atoms that may form a repair.

In the third step, we take into consideration functional dependencies, based on Definition 2.24. We do not present the mapping of key constraints, since they can be expressed by a functional dependency.

In the fourth step, we take into consideration inclusion dependencies, based on Definition 2.25.

The fifth and sixth step, are special cases of the denial constraints, the check constraints and domain constraints respectively, based on Definitions 2.26 and 2.27 respectively.

Definition 5.2 (Extracting Function)**.** *Let P be a database repair problem and $\varphi(P)$ be the corresponding logic program. Assume X is an answer set of $\varphi(P)$. Then, the extracting function α is defined as follows:*

$$\alpha_N(X)=\{N(\bar{c}) \mid p\_keep_N(\bar{c}) \in X\}$$
$$\alpha(X)=\bigcup_{N\in\mathcal{N}} \alpha_N(X)$$

In the previous definition, we collect the set of atoms that will form the new repaired database. They are represented by the atoms with predicate name p_keep_N.

It is also essential to know what are the tuples that form the changes needed to be done (the tuples that have to be deleted or inserted).

Definition 5.3 (Modifications Extracting Function)**.** *Let P be a database repair problem and $\varphi(P)$ be the corresponding logic program. Assume X is an answer set of $\varphi(P)$. Then, let the modifications extracting function Δ be defined as follows:*

$$\Delta^d(X)=\{to\_delete(x_0) \mid delete(x_0) \in X\}$$
$$\Delta^i(X)=\{to\_insert(x_0) \mid insert(x_0) \in X\}$$
$$\Delta(X)=\Delta^d(X) \cup \Delta^i(X)$$

Lemma 5.4. *Let $P = \langle D, E, IC_1\rangle$ be a database repair problem, where $D = \langle I_D, IC\rangle$ and $E = \langle I_E, \emptyset\rangle$, and let $\varphi(P)$ be the corresponding logic program. Let X be an answer set of $\varphi(P)$. Then, $\alpha(X) \subseteq \beta_t(I_D) \cup \beta_t(I_E)$.*

Proof of Lemma 1. Let $P = \langle D, E, IC_1\rangle$, be a database repair problem, where $D = \langle I_D, IC\rangle$ and $E = \langle I_E, \emptyset\rangle$. Also, let $\varphi(P)$ be the corresponding logic program. Let X be an answer set of $\varphi(P)$. Recall from definition 2.43 that, in order for X to be an answer set of $\varphi(P)$, it must be a minimal model of the reduct:

$$\frac{\Pi}{I} = \{H(r) \leftarrow B^+(r) \mid r \in \Pi \wedge B^-(r) \cap I \neq \emptyset\},$$

where $\Pi = ground(\varphi(P))$.

In order to prove that $\alpha(X) \subseteq \beta_t(I_D) \cup \beta_t(I_E)$, assume that $N(\bar{c})$ is an arbitrary atom of $\alpha(X)$. We want to show that it also belongs to $\beta_t(I_D) \cup \beta_t(I_E)$. By the definition of the extracting function (definition 5.2), $p\_keep_N(\bar{c})$ must belong to X (since $\alpha(X)$ is built from X).

Following proposition 2.45, $p\_keep_N(\bar{c})$ is supported by $\frac{\Pi}{I}$ in X, hence there is a rule r in $\frac{\Pi}{I}$ with $p\_keep_N(\bar{c})$ in the head and $B(r) \subseteq X$. Since (5.3c) and (5.4c) are the only rules with $p\_keep_N(\bar{c})$ in the head, we have two distinct cases:

1. If r is of form (5.3c), $p_N(\bar{c})$ needs to be true in X. Following once more proposition 2.45, there is a rule r' in $\frac{\Pi}{I}$ with $p_N(\bar{c})$ in the head such that $B(r') \subseteq X$. Focusing on (5.1), this is the only place where we have rules with predicate p_N in the head. Also, $p_N(\bar{c})$ occurs in $\frac{\Pi}{I}$ only if $N(\bar{c})$ belongs to $\beta_t(I_D)$. This proves that $N(\bar{c}) \in \beta_t(I_D)$.

2. If r is of form (5.4c), $p\_keep^i_N(\bar{c})$ needs to be true in X. Following once more proposition 2.45, there is a rule r' in $\frac{\Pi}{I}$ with $p\_keep^i_N(\bar{c})$ in the head such that $B(r') \subseteq X$. Focusing on (5.2), this is the only place where we have rules with predicate $p\_keep^i_N(\bar{c})$ in the head. Also, $p\_keep^i_N(\bar{c})$ occurs in $\frac{\Pi}{I}$ only if $N(\bar{c})$ belongs to $\beta_t(I_E)$. This proves that $N(\bar{c}) \in \beta_t(I_E)$. □

Lemma 5.5. *Let $P = \langle D, E, IC_1\rangle$ be a database repair problem, where $D = \langle I, IC\rangle$ and $E = \langle E, \emptyset\rangle$. Let $\varphi(P)$ be the corresponding logic program. If X is an answer set of program $\varphi(P)$, then $\alpha(X) \models IC \cup IC_1$.*

Proof of Lemma 2. We want to prove that if X is an answer set of $\varphi(P)$, then $\alpha(X) \models IC \cup IC_1$. By contraposition reasoning, we will prove that if $\alpha(X) \not\models IC \cup IC_1$, then X is not an answer set of $\varphi(P)$. Since $\alpha(X) \not\models IC \cup IC_1$, then $\exists_{\gamma \in IC \cup IC_1}\ \alpha(X) \not\models \gamma$. Since $\alpha(X)$ is obtained from X, according to definition 5.2, we show that there must exist a mapping of γ that is being violated in X. Also, recall from definition 2.43 that, in order for X to be an answer set of $\varphi(P)$, it must be a minimal model of the reduct:

$$\frac{\Pi}{I} = \{H(r) \leftarrow B^+(r) \mid r \in \Pi \wedge B^-(r) \cap I \neq \emptyset\},$$

where $\Pi = ground(\varphi(P))$.

We now consider four distinct cases, based on the type of the integrity constraint of γ.

- **Functional Dependencies** Let $\gamma = FD(N, A, B)$, as defined in 2.24. In order to $\alpha(X) \not\models \gamma$, there must exist two sequences of constants $\bar{c}$ and $\bar{d}$, such that

$$\left[N(\bar{c}) \wedge N(\bar{d}) \wedge \bigwedge_{i \in \#^N_A} c_i = d_i \wedge \bigvee_{j \in \#^N_B} c_j \neq d_j \right]$$

holds. Then, from the previous reasoning, there must exist a j, which we will denote j_0, such that $c_{j_0} \neq d_{j_0}$. By the definition of the extracting function, Definition 5.2, $p\_keep_N(\bar{c})$ and $p\_keep_N(\bar{d})$ must belong to X (since $\alpha(X)$ is built from X). Then, the following rule $r \in \frac{\Pi}{X}$, based on the transformation function ($FD^M(N, A, B)$):

$$\bot \leftarrow p\_keep(\bar{c}) \wedge p\_keep(\bar{d}) \wedge \bigwedge_{i \in \#^N A} c_i = d_i \wedge c_{j_0} = d_{j_0}.$$

will be satisfied in X. If so, inconsistency will be generated. It follows that if $\alpha(X) \not\models \gamma$, then X cannot be an answer set of $\varphi(P)$.

- **Inclusion Dependencies** Let $\gamma = INC(N_1, N_2, A, B)$, as defined in 2.25. In order to $\alpha(X) \not\models \gamma$, there must exist one sequence of constants $\bar{c}$, such that

$$\left[N_1(\bar{c}) \wedge \forall_{\bar{y}} [\neg N_2(\bar{y}) \vee \bigvee_{i=1}^{k} c_{\#_{\kappa_i}^{N_1}} \neq y_{\#_{\varepsilon_i}^{N_2}}] \right]$$

holds. If X is an answer set of $\varphi(P)$, it is a minimal model, under set inclusion, of $\frac{\Pi}{X}$. Then, for all rules r that are reducts of a ground rule r'(where the default literals are satisfied) of a rule $r'' \in \varphi(P)$, X satisfies r. Then, if r is a rule, based on $IND^M(N_1, N_2, A, B)b$, is of the following type:

$$\bot \leftarrow p\_keep_{n_1}(\bar{c})$$

where all default literals are satisfied in r', either $p\_keep_{N_1}(\bar{c}) \notin X$ (which cannot be the case, since, by the definition of the extracting function, Definition 5.2, $p\_keep_{N_1}(\bar{c})$ must belong to X, since $\alpha(X)$ is built from X), or $aux^{A,B}_{N_1,N_2}(\bar{c})$ is true in X. Then, in order for $aux^{A,B}_{N_1,N_2}(\bar{c})$ to be true in X, by the definition of support and rule $IND^M(N_1, N_2, A, B)a$, there must exist a sequence of constants $\bar{d}$ such that:

$$p\_keep_{N_1}(\bar{c}) \wedge p\_keep_{N_2}(\bar{d}) \wedge \bigwedge_{i=1}^{k} c_{\#_{\kappa_i}^{N_1}} \wedge d_{\#_{\varepsilon_i}^{N_2}}$$

is true in X. Then, and once more according to the definition of the extracting function (definition 5.2), we have that

$$N_1(\bar{c}) \wedge N_2(\bar{d}) \wedge \bigwedge_{i=1}^{k} c_{\#_{\kappa_i}^{N_1}} = d_{\#_{\varepsilon_i}^{N_2}}$$

is true in $\alpha(X)$, which is a contradiction with the initial assumption. Therefore, it follows that if $\alpha(X) \not\models \gamma$, then X cannot be an answer set of $\varphi(P)$.

- **Check Constraints** Let $\gamma = CC(N, \kappa, \theta, V)$, as defined in 2.26. In order to $\alpha(X) \not\models \gamma$,

there must exist a sequence of constants $\bar{c}$ such that

$$N(\bar{c}) \wedge c_{\#_\kappa^N} \theta V$$

holds. By the definition of the extracting function, Definition 5.2, $p\_keep_N(\bar{c})$ must belong to X, since $\alpha(X)$ is built from X. Then, the following rule $r \in \frac{\Pi}{X}$, based on the transformation function ($CC^M(N, \kappa, \theta, V)$):

$$\bot \leftarrow p\_keep_N(\bar{c}) \wedge c_{\#_\kappa^N} \theta V$$

will be satisfied. If so, inconsistency will be generated. It follows that if $\alpha(X) \not\models \gamma$, then X cannot be an answer set of $\varphi(P)$.

- **Domain Constraints** Let $\gamma = DoC(N, \kappa, Do)$, as defined in 2.27. In order to $\alpha(X) \not\models \gamma$, there must exist a sequence of constants $\bar{c}$, such that:

$$\left[N(\bar{c}) \wedge \bigwedge_{i=1}^{k} c_{\#_\kappa^N} \neq val_i \right]$$

holds. If X is an answer set of $\varphi(P)$, it is a minimal model, under set inclusion, of $\frac{\Pi}{X}$. Then, for all rules r that are reducts of a ground rule r'(where the default literals are satisfied) of a rule $r'' \in \varphi(P)$, X satisfies r. Then, if r is a rule, based on $DoC^M(N, \kappa, D)b$, of the following type:

$$\bot \leftarrow p\_keep_N(\bar{c})$$

where all default literals are satisfied in r', either $p\_keep_N(\bar{c}) \notin X$ (which cannot be the case, since, by the definition of the extracting function, Definition 5.2, $p\_keep_N(\bar{c})$ must belong to X, since $\alpha(X)$ is built from X), or $d_N^\kappa(c_{\#_\kappa^N})$ is true in X. Then, in order for $d_N^\kappa(c_{\#_\kappa^N})$ to be true in X, by the definition of support and rule ($DoC^M(N, \kappa, Do)a$), there must exist a $val \in Do$ such that

$$val = c_{\#_\kappa^N}$$

is true in X. Then, and once more according to the definition of the extracting function (definition 5.2), we have that

$$N(\bar{c}) \wedge val = c_{\#_\kappa^N}$$

is true in $\alpha(X)$, which is a contradiction with the initial assumption. Therefore, it follows that if $\alpha(X) \not\models \gamma$, then X cannot be an answer set of $\varphi(P)$. □

Corollary 5.6 (Repair Soundness). *Let $P = \langle D, E, IC_1 \rangle$ be a database repair problem. Let $\varphi(P)$ be the corresponding logic program. If X is an answer set of program $\varphi(P)$, $\alpha(X)$ is a*

repair of D with respect to P.

Proof. Immediately follows from lemmas 5.4, 5.5. □

Lemma 5.7 (Completeness)**.** *Let $P = \langle D, E, IC_1 \rangle$ be a database repair problem. Let $\varphi(P)$ be the corresponding logic program. Let D' be a repair of D with respect to P. Then there must exist an answer set X of $\varphi(P)$ such that $\beta_t(I'_D) = \alpha(X)$.*

Proof of Lemma 3. Let P be a database repair problem, regarding databases $D = \langle I_D, IC \rangle$ and $E = \langle I_E, \emptyset \rangle$, and a set of integrity constraints, IC_1. Let $\varphi(P)$ be the corresponding logic program. Let $D' = \langle I'_D, IC \cup IC_1 \rangle$ be a repair of D with respect to P. Then, there must exist an answer set X of $\varphi(P)$ such that $\beta_t(I'_D) = \alpha(X)$.

We consider the following set of atoms X:

$$
\begin{aligned}
X = &\{p_N(\bar{c}) \mid N(\bar{c}) \in \beta_t(I_D)\} \\
&\cup \{p\_keep^i_N(\bar{x}) \mid N(\bar{c}) \in \beta_t(I_E)\} \\
&\cup \{delete(c_0) \mid N(\bar{c}) \in \beta_t(I_D) \wedge N(\bar{c}) \notin \beta_t(I'_D)\} \\
&\cup \{n\_delete(c_0) \mid N(\bar{c}) \in \beta_t(I_D) \wedge N(\bar{c}) \in \beta_t(I'_D)\} \\
&\cup \{insert(c_0) \mid N(\bar{c}) \in \beta_t(I_E) \wedge N(\bar{c}) \in \beta_t(I'_D)\} \\
&\cup \{n\_insert(c_0) \mid N(\bar{c}) \in \beta_t(I_E) \wedge N(\bar{c}) \notin \beta_t(I'_D)\} \\
&\cup \{aux^{A,B}_{N_1,N_2}(\bar{c}) \mid INC(N_1, N_2, A, B) \in IC \cup IC_1 \wedge N_1(\bar{c}) \in \beta_t(I'_D) \wedge \\
&\qquad \exists_{\bar{d}} \left[N_2(\bar{d}) \in \beta_t(I'_D) \wedge \bigwedge_{i=1}^{k} c_{\#^{N_1}_{\kappa_i}} = c_{\#^{N_2}_{\varepsilon_i}} \right]\} \\
&\cup \{d^{\kappa}_N(val) \mid DoC(N, \kappa, Do) \in IC \cup IC_1 \wedge val \in Do\} \\
&\cup \{p\_keep_N(\bar{c}) \mid N(\bar{c}) \in \beta_t(I'_D)\}
\end{aligned}
$$

For X being an answer set of $\varphi(P)$ such that $\beta_t(I'_D) = \alpha(X)$, we need to verify that X is a minimal model, under set inclusion, of:

$$\frac{\Pi}{X} = \{H(r) \leftarrow B^+(r) \mid r \in ground(\Pi) \wedge B^-(r) \cap X \neq \emptyset\}$$

where $\Pi = ground(\varphi(P))$. We shall start to prove that X is a model of $\frac{\Pi}{X}$.

In order for X to be a model of $\frac{\Pi}{X}$, for every rule r in $\frac{\Pi}{X}$, $B(r) \subseteq X \Rightarrow H(r) \in X$. From here on, let us assume that, for every rule r, $r \in \Pi$, such that r is a reduct of a ground rule r', of a rule $r'' \in \varphi(P)$. We shall also assume that $B(r) \subseteq X$.

If r'' is of the form (5.1), then we must prove that $p_N(\bar{c}) \in X$. As we can see, rule r'' is only added to $\varphi(P)$ if $N(\bar{c}) \in \beta_t(I_D)$. By the definition of X, $p_N(\bar{c}) \in X$ if $N(\bar{c}) \in \beta_t(I_D)$, as we wanted to show.

If r'' is of the form (5.2), then we must prove that $p\_keep^i_N(\bar{x}) \in X$. As we can see, rule r'' is only added to $\varphi(P)$ if $N(\bar{c}) \in \beta_t(I_E)$. By the definition of X, $p\_keep^i_N(\bar{c}) \in X$ if $N(\bar{c}) \in \beta_t(I_E)$, as we wanted to show.

If r'' is a rule of the type (5.3a), r is of the form:

$$delete(c_0) \leftarrow p_N(c_0, c_1, ..., c_n).$$

Then, we must prove that $delete(c_0) \in X$. Notice that since $r \in \frac{\Pi}{X}$, all default literals from $B(r')$ are satisfied in X. In this case, it means that $n\_delete(c_0) \notin X$. By the definition of X, $delete(c_0) \in X$ if $N(c_0, c_1, ..., c_n) \in \beta_t(I_D) \wedge N(c_0, c_1, ..., c_n) \notin \beta_t(I'_D)$. Since we assumed that the body is true , $p_N(c_0, c_1, ..., c_n) \in X$, which means that $N(c_0, c_1, ..., c_n) \in \beta_t(I_D)$. It is still left to prove that $N(c_0, c_1, ..., c_n) \notin \beta_t(I'_D)$. Since $n\_delete(x_0) \notin X$, then, either $N(c_0, c_1, ..., c_n) \notin \beta_t(I_D)$ or $N(c_0, c_1, ..., c_n) \notin \beta_t(I'_D)$. Since we already proved that $N(c_0, c_1, ..., c_n) \in \beta_t(I_D)$, it follows that $N(c_0, c_1, ..., c_n) \notin \beta_t(I'_D)$, as we wanted to prove.

If r'' is a rule of the type (5.3b), r is of the form:

$$n\_delete(c_0) \leftarrow p_N(c_0, c_1, ..., c_n).$$

Then, we must prove that $n\_delete(c_0) \in X$. Notice that since $r \in \frac{\Pi}{X}$, all default literals from $B(r')$ are satisfied in X. In this case, it means that $delete(c_0) \notin X$. By the definition of X, $n\_delete(c_0) \in X$ if $N(c_0, c_1, ..., c_n) \in \beta_t(I_D) \wedge N(c_0, c_1, ..., c_n) \in \beta_t(I'_D)$. Since we assumed that the body is true , $p_N(c_0, c_1, ..., c_n) \in X$, which means that $N(c_0, c_1, ..., c_n) \in \beta_t(I_D)$. It is still left to prove that $N(c_0, c_1, ..., c_n) \in \beta_t(I'_D)$. Since $delete(c_0) \notin X$, then, either $N(c_0, c_1, ..., c_n) \notin \beta_t(I_D)$ or $N(c_0, c_1, ..., c_n) \in \beta_t(I'_D)$. Since we already proved that $N(c_0, c_1, ..., c_n) \in \beta_t(I_D)$, it follows that $N(c_0, c_1, ..., c_n) \in \beta_t(I'_D)$, as we wanted to prove.

If r'' is a rule of the type (5.3c), r is of the form:

$$p\_keep_N(c_0, c_1, ..., c_n) \leftarrow p_N(c_0, c_1, ..., c_n).$$

Then, we must prove that $p\_keep_N(c_0, c_1, ..., c_n) \in X$. Notice that, since $r \in \frac{\Pi}{X}$, all default literals from $B(r')$ are satisfied in X. In this case, it means that $delete(c_0) \notin X$. By the definition of X, $p\_keep_N(c_0, c_1, ..., c_n) \in X$ if $N(c_0, c_1, ..., c_n) \in \beta_t(I'_D)$. Since we assumed that the body is true, $p_N(c_0, c_1, ..., c_n) \in X$, which means that $N(x_0, x_1, ..., x_n) \in \beta_t(I_D)$. It is still left to prove that $N(c_0, c_1, ..., c_n) \in \beta_t(I'_D)$. Since $delete(c_0) \notin X$, then, either $N(c_0, c_1, ..., c_n) \notin \beta_t(I_D)$ or $N(c_0, c_1, ..., c_n) \in \beta_t(I'_D)$. Since we already proved that $N(c_0, c_1, ..., c_n) \in \beta_t(I_D)$, it follows that $N(c_0, c_1, ..., c_n) \in \beta_t(I'_D)$, as we wanted to prove.

If r'' is a rule of the type (5.4a), r is of the form:

$$insert(c_0) \leftarrow p\_keep^i_N(c_0, c_1, ..., c_n).$$

Then, we must prove that $insert(c_0) \in X$. Notice that, since $r \in \frac{\Pi}{X}$, all default literals from $B(r')$are satisfied in X. In this case, it means that $n\_insert(c_0) \notin X$. By the definition of X, $insert(c_0) \in X$ if $N(c_0, c_1, ..., c_n) \in \beta_t(I_E) \wedge N(c_0, c_1, ..., c_n) \in \beta_t(I'_D)$. Since we assumed that the body is true, $p\_keep^i_N(c_0, c_1, ..., c_n) \in X$, which means that

$N(c_0, c_1, ..., c_n) \in \beta_t(I_E)$. It is still left to prove that $N(c_0, c_1, ..., c_n) \in \beta_t(I'_D)$. Since $n\_insert(c_0) \notin X$, then, either $N(c_0, c_1, ..., c_n) \notin \beta_t(I_E)$ or $N(c_0, c_1, ..., c_n) \in \beta_t(I'_D)$. Since we already proved that $N(c_0, c_1, ..., c_n) \in \beta_t(I_E)$, it follows that $N(c_0, c_1, ..., c_n) \in \beta_t(I'_D)$, as we wanted to prove.

If r'' is a rule of the type (5.4b), r is of the form:

$$n\_insert(c_0) \leftarrow p\_keep^i_N(c_0, c_1, ..., c_n).$$

Then, we must prove that $n\_insert(c_0) \in X$. Notice that, since $r \in \frac{\Pi}{X}$, all default literals from $B(r')$ are satisfied in X. In this case, it means that $insert(c_0) \notin X$. By the definition of X, $n\_insert(c_0) \in X$ if $N(c_0, c_1, ..., c_n) \in \beta_t(I_E) \wedge N(c_0, c_1, ..., c_n) \notin \beta_t(I'_D)$. Since we assume that the body is true, $p\_keep^i_N(c_0, c_1, ..., c_n) \in X$, which means that $N(c_0, c_1, ..., c_n) \in \beta_t(I_E)$. It is still left to prove that $N(c_0, c_1, ..., c_n) \notin \beta_t(I'_D)$. Since $insert(c_0) \notin X$, then, either $N(c_0, c_1, ..., c_n) \notin \beta_t(I_E)$ or $N(c_0, c_1, ..., c_n) \notin \beta_t(I'_D)$. Since we already proved that $N(c_0, c_1, ..., c_n) \in \beta_t(I_E)$, it follows that $N(c_0, c_1, ..., c_n) \notin \beta_t(I'_D)$, as we wanted to prove.

If r'', based on rule (5.4c), r is of the form:

$$p\_keep_N(c_0, c_1, ..., c_n) \leftarrow insert(c_0) \wedge p\_keep^i_N(c_0, c_1, ..., c_n).$$

Then, we must prove that $p\_keep_N(\bar{c}) \in X$. By the definition of X, $p\_keep_N(\bar{c}) \in X$ if $N(c_0, c_1, ..., c_n) \in \beta_t(I'_D)$. Since we assume the body is true, $insert(c_0) \in X$, which means that $N(c_0, c_1, ..., c_n) \in \beta_t(I_E) \wedge N(c_0, c_1, ..., c_n) \in \beta_t(I'_D)$. It immediately follows that $N(c_0, c_1, ..., c_n) \in \beta_t(I_E) \wedge N(c_0, c_1, ..., c_n) \in \beta_t(I'_D)$, as we wanted to prove.

If r'' is a rule of the form ($FD^M(N, A, B)$), mapped from an integrity constraint $FD(N, A, B)$, from definition 2.24, we will prove that, if $B(r) \subseteq X$, then D' is not a repair, contrary to the assumption. If we assume that the body is true, then $p\_keep_N(\bar{c}) \in X$ and $p\_keep_N(\bar{d}) \in X$, such that $\bigwedge_{i \in \#^N_A} c_i = d_i \wedge c_j \neq d_j$, for some $j \in \#^N_B$. According to the definition of X, it follows that $N(\bar{c}) \in \beta_t(I'_D) \wedge N(\bar{d}) \in \beta_t(I'_D)$, such that $\bigwedge_{i \in \#^N A} c_i = d_i \wedge c_j \neq d_j$, which is a contradiction considering the definition 2.24 of a functional dependency. Given this, it follows that $\beta_t(I'_D) \not\models FD(N, A, B)$. In the sequel, D' is not a repair. Then, contrary to the assumption, $B(r) \nsubseteq X$, so X satisfies r.

Since keys are a specific case of the functional dependencies, we shall not considerate that case, since it has already been proven for a more general case.

If r'' is a rule of the form ($IND^M(N_1, N_2, A, B)a$), then we must prove that $aux^{A,B}_{N_1,N_2}(\bar{c}) \in X$. If we assume that the body is true, then $p\_keep_{N_1}(c_0, c_1, ..., c_n) \in X$ and $p\_keep_{N_2}(d_0, d_1, ..., d_n) \in X$, such that $\bigwedge_{i=1}^{k} c_{\#^{N_1}_{\kappa_i}} = d_{\#^{N_2}_{\varepsilon_i}}$. As we can see, the rule r is only added to Π if $INC(N_1, N_2, A, B) \in IC \cup IC_1$. By the definition of X, $aux^{A,B}_{N_1,N_2}(\bar{c}) \in X$ if $INC(N_1, N_2, A, B) \in IC \cup IC_1 \wedge N_1(\bar{c}) \in \beta_t(I'_D) \wedge N_2(\bar{d}) \in \beta_t(I'_D)$. Since the body is true, it follows that $N_1(\bar{c}) \in \beta_t(I'_D) \wedge N_2(\bar{d}) \in \beta_t(I'_D)$, as we wanted to prove.

If r'', is a rule of the type ($IND^M(N_1, N_2, A, B)b$), r is of the form:

$$\bot \leftarrow p\_keep_{N_1}(\bar{c}),$$

where all default literals from $B(r')$ are satisfied in X, mapped from an integrity constraint $INC(N_1, N_2, A, B)$, from definition 2.25. We will prove that, if $B(r) \subseteq X$, then D' is not a repair, contrary to the assumption. If we assume that the body is true, then $p\_keep_{N_1}(\bar{c}) \in X$. From the definition of X, if follows that $N_1(\bar{c}) \in \beta_t(I'_D)$. Also, $aux^{A,B}_{N_1,N_2}(\bar{c}) \notin X$. Then, by the definition of X, either $INC(N_1, N_2, A, B) \notin IC \cup IC_1$ (which cannot be the case, otherwise the rule would never exist), or $N_1(\bar{c}) \notin \beta_t(I'_D)$ (which cannot be the case as well, since we already proved otherwise) or

$$\forall_{\bar{d}} \left[N_2(\bar{d}) \notin \beta_t(I'_D) \vee \bigvee_{i=1}^{k} c_{\#^{N_1}_{\kappa_i}} \neq d_{\#^{N_2}_{\varepsilon_i}} \right],$$

which is the case, since $aux^{A,B}_{N_1,N_2}(\bar{c}) \notin X$. Then, there is a contradiction with respect to definition 2.25. Given this, it follows that $\beta_t(I'_D) \not\models INC(N_1, N_2, A, B)$. In the sequel, D' is not a repair. Then, contrary to the assumption, $B(r) \nsubseteq X$, so X satisfies r.

If r'' is a rule of the form ($CC^M(N, \kappa, \theta, V)$), mapped from an integrity constraint $CC(N, \kappa, \theta, V)$, from definition 2.26, we will prove that, if $B(r) \subseteq X$, then D' is not a repair, contrary to the assumption. If we assume that the body is true, then $p\_keep_N(\bar{c}) \in X$, such that $c_{\#^N_\kappa}\theta V$. According to the definition of X, it follows that $N(\bar{c}) \in \beta_t(I'_D)$, such that $c_{\#^N_\kappa}\theta V$, which is a contradiction considering the definition (2.26) of a check constraint. Given this, it follows that $\beta_t(I'_D) \not\models CC(N, \kappa, \theta, V)$. In the sequel, D' is not a repair. Then, contrary to the assumption, $B(r) \nsubseteq X$, so X satisfies r.

If r'' is of the form ($DoC^M(N, \kappa, Do)a$), then, we must prove that $d^\kappa_N(val) \in X$. As we can see, rule r is only added to Π if $DoC(N, \kappa, Do) \in IC \cup IC_1$. If such integrity constraint exists, then, for every $val \in Do$, we add the rule $d^\kappa_N(val)$ to Π. By the definition of X, $d^\kappa_N(val) \in X$ if $DoC(N, \kappa, Do) \in IC \cup IC_1 \wedge val \in Do$, as we wanted to prove.

If r'' is a rule of the type ($DoC^M(N, \kappa, D)b$), r if of the form:

$$\bot \leftarrow p\_keep_N(\bar{c}),$$

where all default literals from $B(r')$ are satisfied in X, mapped from an integrity constraint $DoC(N, \kappa, Do)$, from definition 2.27. We will prove that, if $B(r) \subseteq X$, then D' is not a repair, contrary to the assumption. If we assume that the body is true, then $p\_keep_N(\bar{c}) \in X$. Then, by the definition of X, $N(\bar{c}) \in \beta_t(I'_D)$. Also, $d^\kappa_N(c_{\#^N_\kappa}) \notin X$. Then, from the definition of X, either $DoC(N, \kappa, Do) \notin IC \cup IC_1$ (which cannot be the case, otherwise the rule would never exist), or:

$$\forall_{val \in Do} val \neq c_{\#^N_\kappa},$$

which is the case, since $d_N^{\kappa}(c_{\#_N^{\kappa}}) \notin X$. Then, there is a contradiction with respect to definition 2.27. Given this, it follows that $\beta_t(I'_D) \not\models DoC(N, \kappa, Do)$. In the sequel, D' is not a repair. Then, contrary to the assumption, $B(r) \nsubseteq X$, so X satisfies r.

We now prove that X is a minimal model of $\frac{\Pi}{X}$. In order to do it, let's assume that there is a model Y of $\frac{\Pi}{X}$, such that $Y \subseteq X$.

Let us pick an arbitrary atom p from X of the form $p_N(\bar{c})$. If follows that $N(\bar{c}) \in \beta_t(I_D)$. In order for $p \in Y$, we will show that there is a rule $r \in \frac{\Pi}{X}$ such that $H(r) = p \wedge B(r) \subseteq Y$. Let r be then a rule of the form (5.1). Since the body is the empty set and, it is trivially contained in Y. Therefore, since Y is a model of $\frac{\Pi}{X}$, $p_N(\bar{x}) \in Y$.

Let us pick an arbitrary atom p from X of the form $p\_keep_N^i(\bar{c})$. If follows that $N(\bar{c}) \in \beta_t(I_E)$. In order for $p \in Y$, we will show that there is a rule $r \in \frac{\Pi}{X}$ such that $H(r) = p \wedge B(r) \subseteq Y$. Let r be then a rule of the form (5.2). Since the body is the empty set, it is trivially contained in Y. Therefore, since Y is a model of $\frac{\Pi}{X}$, $p\_keep_N^i(\bar{x}) \in Y$.

Let us pick an arbitrary atom from X of the form $delete(c_0)$. If follows that $N(\bar{c}) \in \beta_t(I_D)$. Then, by the definition of X, $p_N(\bar{c}) \in X$. As we saw before, $p_N(\bar{c}) \in X \Rightarrow p_N(\bar{c}) \in Y$. In order for $p \in Y$, we will show that there is a rule $r \in \frac{\Pi}{X}$ such that $H(r) = p \wedge B(r) \subseteq Y$. Let r be then a rule of the form (5.3a). If we take a look at rules (5.3a) and (5.3b), if an atom $p_N(\bar{c})$ is true in X, then we must have the atom $delete(c_0)$ true in X or $n\_delete(c_0)$ true in X (but not both). Since we assumed that $Y \subseteq X$, if $n\_delete(c_0) \notin X$, $n\_delete(c_0) \notin Y$, otherwise, $Y \nsubseteq X$. Therefore, since Y is a model of $\frac{\Pi}{X}$, $delete(c_0) \in Y$.

The same line of reasoning is applied when we consider an arbitrary atom from X of the form $n\_delete(c_0)$.

Let us pick an arbitrary atom from X of the form $insert(c_0)$. If follows that $N(\bar{c}) \in \beta_t(I_E)$. Then, by the definition of X, $p\_keep_N^i(\bar{c}) \in X$. As we saw before, $p\_keep_N^i(\bar{c}) \in X \Rightarrow p\_keep_N^i(\bar{c}) \in Y$. In order for $p \in Y$, we will show that there is a rule $r \in \frac{\Pi}{X}$ such that $H(r) = p \wedge B(r) \subseteq Y$. Let r be then a rule of the form (5.4a). If we take a look at rules (5.4a) and (5.4b), if an atom $p\_keep_N^i(\bar{c})$ is true in X, then we must have the atom $insert(c_0)$ true in X or $n\_insert(c_0)$ true in X (but not both). Since we assumed that $Y \subseteq X$, if $n\_insert(c_0) \notin X$, $n\_insert(c_0) \notin Y$, otherwise, $Y \nsubseteq X$. Therefore, since Y is a model of $\frac{\Pi}{X}$, $insert(c_0) \in Y$.

The same line of reasoning is applied when we consider an arbitrary atom from X of the form $n\_insert(c_0)$.

Let us pick an arbitrary atom p from X of the form $p\_keep_N(\bar{c})$. By the definition of a repair and the definition of X, $(N(\bar{c}) \in \beta_t(I_D) \vee N(\bar{c}) \in \beta_t(I_E))$. Then, either $p_N(\bar{c}) \in X$ or $p\_keep_N^i(\bar{c}) \in X$. As we saw before, $p_N(\bar{c}) \in X \Rightarrow p_N(\bar{c}) \in Y$ and, as well, $p\_keep_N^i(\bar{c}) \in X \Rightarrow p\_keep_N^i(\bar{c}) \in Y$. In order for $p \in Y$, we will show that there is a rule $r \in \frac{\Pi}{X}$ such that $H(r) = p \wedge B(r) \subseteq Y$. There are two rules such that the previous condition holds.

- Let r be then a rule of the form (5.3c). We have already proved that $p_N(\bar{c}) \in Y$. We have also seen, before, that $delete(c_0) \notin X \Rightarrow delete(c_0) \notin Y$, otherwise $Y \nsubseteq X$. Therefore, since Y is a model of $\frac{\Pi}{X}$, $p\_keep_N(\bar{c}) \in Y$.

- Let r be then a rule of the form (5.4c). We have already proved that $p\_keep^{i}_{N}(\bar{c}) \in Y$. We have also seen, before, that $insert(c_0) \in X \Rightarrow insert(c_0) \in Y$, otherwise $Y \nsubseteq X$. Therefore, since Y is a model of $\frac{\Pi}{X}$, $p\_keep(\bar{c}) \in Y$.

Let us pick an arbitrary atom p from X of the form $aux^{A,B}_{N_1,N_2}(\bar{c})$. If follows that $p\_keep_{N_1}(\bar{c}) \in X$ and $p\_keep_{N_1}(\bar{c}) \in X$. As we saw before, $p\_keep_N(\bar{c}) \in X \Rightarrow p\_keep_N(\bar{c}) \in Y$. Then, $p\_keep_{N_1}(\bar{c}) \in Y$ and $p\_keep_{N_1}(\bar{c}) \in Y$. In order for $p \in Y$, we will show that there is a rule $r \in \frac{\Pi}{X}$ such that $H(r) = p \wedge B(r) \subseteq Y$. Let r be then a rule of the form ($IND^M(N_1, N_2, A, B)a$). Since we have already seen that the atoms exist in Y, the body is satisfied in Y. Therefore, since Y is a model of $aux^{A,B}_{N_1,N_2}(\bar{c}) \in Y$.

Let us pick an arbitrary atom p from X of the form $d^{\kappa}_{N}(val)$. If follows that $DoC(N, \kappa, Do) \in IC \cup IC_1 \wedge val \in Do$. In order for $p \in Y$, we will show that there is a rule $r \in \frac{\Pi}{X}$ such that $H(r) = p \wedge B(r) \subseteq Y$. Let r be then a rule of the form (5.6). Since the body is the empty set and, it is trivially contained in Y. Therefore, since Y is a model of $\frac{\Pi}{X}$, $d^{\kappa}_{N}(val) \in Y$.

We have shown that $X \subseteq Y$. Following the initial assumption that $Y \subseteq X$, we conclude that $X = Y$.

We have now investigated all rules in $\frac{\Pi}{X}$ their instances are satisfied by X. Furthermore, we checked, for all atoms in X, that they cannot be excluded in any model $Y \subset X$ of $\frac{\Pi}{X}$.

It is still left to prove that $\beta_t(I'_D) = \alpha(X)$. According to definition 5.1, the repairs are built with the atoms $p\_keep_N(\bar{c})$, that are drawn from X. If we take a look at the definition of X, $p\_keep_N(\bar{c}) \in X$ if $N(\bar{c}) \in \beta_t(I'_D)$. It is then clear that $\beta_t(I'_D) = \alpha(X)$. □

Corollary 5.8 (Repair Completeness)**.** *Let P be a database repair problem, regarding database $D = \langle I_D, IC\rangle$ and the set of integrity constraints IC_1. Let $\varphi(P)$ be the corresponding logic program. If $D' = \langle I'_D, IC \cup IC_1\rangle$ is a repair, then there is an answer set X of $\varphi(P)$, such that $\beta_t(I'_D) = \alpha(X)$.*

Proof. Immediately follows from lemmas 5.4, 5.7. □

Corollary 5.9 (Repair Soundness and Completeness)**.** *Let P be a database repair problem, regarding database $D = \langle I_D, IC\rangle$ and the set of integrity constraints IC_1. Let $\varphi(P)$ be the corresponding logic program. Then, $\alpha(X)$ is a repair of X if and only if X is an answer set of $\varphi(P)$.*

Proof. Immediately follows from lemmas 5.4, 5.5, 5.7. □

5.2 Minimality Statements

So far, we have only considered a general approach, considering only the more general definition of a repair. Here, we will take into consideration minimality under cardinality of operations and minimality under set inclusion as well. We divide these approaches

and introduce a new transformation function for each one of them, in order to build a new logic program that takes into account these minimality statements. We begin first with the minimality under cardinality, and then we go into minimality under set inclusion.

5.2.1 Cardinality Distance

Minimal repairs under cardinality of operations are the repairs that are obtained by performing as little operations possible in the original database. It is an adequate minimality criteria, since we are dealing with relational databases. Therefore, it is very important to consider the number of "changes" that will occur during the whole process.

In order to encode this minimality into a logic program, let us define a new transformation function, based on the previously defined one, in Definition 5.1.

Definition 5.10 (Problem Transformation Function - Cardinality Based)**.** *Let $P = \langle D, E, IC_1 \rangle$ be a database repair problem, where $D = \langle I_D, IC \rangle$, and $E = \langle I_E, \emptyset \rangle$. Let $\varphi^C(P)$ represent the transformation function, with respect to P, regarding minimality under cardinality, defined as $\varphi(P) \cup card(P)$, where $card(P)$ is defined as follows:*

$$count(M) \leftarrow M = \#sum[delete(DRowid), insert(IRowid)]. \tag{5.7}$$

$$\#minimize[count(M) = M]. \tag{5.8}$$

Since we introduced a new transformation function, we extend Definitions 5.2 and 5.3 to assume, in this particular case, that X is an answer set of $\varphi^C(P)$.

In this approach, we explore the potential of the answer set grounder and solver, gringo and clasp respectively. As we can see, we use two non-standard instructions, the $\#sum$ (5.7) and the $\#minimize$ (5.8). Following the manual of clasp[1], $\#sum$ is an aggregate, which is an operation on a multi-set of weighted literals that evaluates to some value. In combination with comparisons, we can extract a truth value from an aggregate's evaluation, thus, obtaining an aggregate atom. Aggregate atoms are of the following form:

$$l\text{op}[L_1 = w_1, ..., L_n = w_n]u$$

An aggregate has a lower bound l, an upper bound u, an operation op, and a multiset of literal L_i each assigned to a weight w_i. An aggregate is true if operation op applied to the multiset of weights of true literals is between the bounds. The aggregate $\#sum$, as the name indicates, sums the weights of every literal. In our approach, the weight of each literal is set to 1. According to the previous rules, we are counting the number of atoms that are meant to be inserted and deleted. Afterwards, we wish to minimize that very same number. In order to do that, we use an optimization statement, the $\#minimize$ statement.

The optimization statements extend the basic question of whether a set of atoms is an answer set to whether it is an optimal answer set. In a multiset notation (the one being

[1] http://www.cs.uni-potsdam.de/clasp/

used), weights may be provided. We are then, in (5.6), instantiating the $count(M)$ atom, and associating the value of M as its weight. That way, we want to minimize that very same value. The semantic of an optimization statement is then very intuitive: an answer set is *optimal* if the sum of the weights of literals is maximal or minimal, as required by the statement, among all answer sets of the given program.

For further information about the semantics and syntax of *aggregates* and *optimization statements*, we recommend the full reading of the clasp manual.

With this new program, we are restricting the answer sets of $\varphi^C(P)$. Also, we do not change the the rules of the original transformation function, and we simply add new ones that restrict even more the possible answer sets of a logic program (since we are minimizing the number of operations performed). It is acceptable that all of the previous properties still hold, that is, this new transformation function is still sound and complete. Therefore, all answer sets of P that are generated are minimal repairs under cardinality of operations, with respect to P. Also, we are generating all possible minimal repairs of P, considering minimality under cardinality of operations, if it is the case of being more than one possible minimal repair.

5.2.2 Set Inclusion Distance

According to this kind of minimality, a repair D' of D is minimal under set inclusion if there is no other repair D'' such that $\Delta(D, D'') \subset \Delta(D, D')$, where Δ stands for the symmetric difference.

To address this problem, we used a technique called *saturation*, based on the one presented in [EFLP99], where an algorithm for abductive diagnosis (over disjunctive logic programming) and consistency-based diagnosis was presented. We adapted that algorithm to our specific problem, without using disjunctions. Using this in our approach, we only generate an answer set X of a repair problem P, if there is not another answer set Y of P, such that $\Delta(Y) \subset \Delta(X)$ and $|\Delta(X)| - |\Delta(Y)| = 1$, where $|R|$ denotes the cardinality of set R, and Δ stands for the modifications extracting function. Unfortunately, *saturation* is only able to partially deal with the problem. It may be the case that some repairs, without being minimal under set inclusion, may still be generated. Recall that the complexity classes of database repairing lies between the *NP-hard* complexity class and the Σ_2^p complexity class. Also, answer set programming is oriented towards *NP*-hard problems. Although some answer set solvers provide some optimizations statements that are able to deal with problems of higher complexity, they do not provide any optimization statement that can help in this specific problem. In order to encode this minimality in a logic program, let us define a new transformation function based on the previously defined one, in Definition 5.1.

Definition 5.11 (Problem Transformation Function - Set Inclusion Based)**.** *Let $P = \langle D, E, IC_1\rangle$ be a database repair problem, where $D = \langle I_D, IC\rangle$, and $E = \langle I_E, \emptyset\rangle$. Let $\varphi^S(P)$ represent the transformation function, with respect to P, regarding minimality under set inclusion, defined*

as $\varphi(P) \cup setinc(P)$, where $setinc(P)$ is defined as follows:

1. *Instance Multiplication: For every relation $N \in \{N' \mid \langle N', C, S\rangle \in I_D\}$, we add the following rules to the program:*

$$N\_new(Newid, \bar{x}) \leftarrow p\_keep_N(\bar{x}) \wedge delete(Newid).$$
$$N\_new(x_0, \bar{x}) \leftarrow p_N(\bar{x}) \wedge delete(x_0).$$
$$N\_new(Newid, \bar{x}) \leftarrow p\_keep_N(\bar{x}) \wedge insert(Newid) \wedge Newid \neq x_0.$$

2. *Functional Dependencies/Keys: For every functional dependency $FD(N, A, B) \in IC \cup IC_1$, we add the following rules to the repair program:*

   For each $j \in \#_B^N$ we add a rule of the type:

$$ok(Newid) \leftarrow N\_new(Newid, \bar{x}) \wedge N\_new(Newid, \bar{y}) \wedge \bigwedge_{i \in \#_X^N} x_i = y_i \wedge x_j \neq y_j.$$

3. *Inclusion Dependencies: For every inclusion dependency $IND(N_1, N_2, A, B) \in IC \cup IC_1$, we add the following rules to the repair program:*

$$new\_aux^{A,B}_{N_1,N_2}(Newid, \bar{x}) \leftarrow N_1\_new(Newid, \bar{x}) \wedge N_2\_new(Newid, \bar{y}) \wedge \bigwedge_{i=1}^{k} x_{\#^{N_1}_{\kappa_i}} = y_{\#^{N_2}_{\varepsilon_i}}.$$

$$ok(Newid) \leftarrow N_1\_new(Newid, \bar{x}) \wedge not\ new\_aux^{A,B}_{N_1,N_2}(Newid, \bar{x}).$$

4. *Check Constraints: For every check constraint $CC(N, \kappa, \theta, V) \in IC \cup IC_1$, we add the following rules to the repair program:*

$$ok(Newid) \leftarrow N\_new(Newid, \bar{x}) \wedge x_{\#^N_\kappa}\ \theta\ V.$$

5. *Domain Constraints: For every domain constraint $DoC(N, \kappa, Do) \in IC \cup IC_1$, where $Do = \{val_1, val_2, ..., val_n\}$, we add the following rules to the repair program:*

$$ok(Newid) \leftarrow N\_new(Newid, \bar{x}) \wedge not\ d^\kappa_N(x_{\#^N_\kappa}).$$

6. *Model elimination: At the end, we add the following rules:*

$$\bot \leftarrow not\ ok(Newid), delete(Newid).$$
$$\bot \leftarrow not\ ok(Newid), insert(Newid).$$

Since we introduced a new transformation function, we extend Definitions 5.2 and 5.3 to assume, in this particular case, that X is an answer set of $\varphi^S(P)$.

The idea behind the previous definition is actually quite simple. If X is a potential answer set of $\varphi(P)^S$, for every atom $delete(Rowid) \in X$, we check if $\alpha(X) \cup N(\bar{c})$, where $c_0 = Rowid$ and N is the relation that has that tuple with $Rowid = c_0$, is a repair itself. If so, we discard X, since it is not minimal under set inclusion. The identical is made with the atoms of the form $insert(Rowid)$. We check if $\alpha(X) \setminus N(\bar{c})$, where $c_0 = Rowid$ and N is the relation that has that tuple with $Rowid = c_0$, is a repair itself. If so, we discard X, since, once again, it is not minimal under set inclusion.

With this approach, we eliminate many repairs of $\varphi(P)$ that are not minimal under set inclusion. However, there may still be repairs of $\varphi(P)$ that are not minimal under set inclusion that will still be generated. Notice that we did not change any of the rules of the original transformation function, so, we are only restricting the generated answer sets of $\varphi(P)$. It is easy to see that this approach is still complete. We still generate all possible minimal repairs under set inclusion with respect to P, along with some more that may not be minimal under set inclusion, thus being soundness not achieved.[2]

Now that we have introduced our mapping of a minimal repair problem into a logic program, it is time to present the implementation of our approach, by presenting the developed graphical user interface and its architecture. In the next chapter, we present *DRSys*, a database repair application.

[2]In order to obtain a sound approach, another step would be needed. A possible approach would be to generate a Prolog program and, using a Prolog interpreter, eliminate all repairs of $\varphi(P)$ that are not minimal under set inclusion. That way, we would get soundness as well.

6

Database Repair System - *DRSys*

In this chapter, we describe the "Database Repair System", which, from now on, we shall address to as *DRSys*, the database repair application developed throughout this dissertation. It allows the user to define new integrity constraints at any point in time, and, if they lead the database to an inconsistent state, it computes database repairs, having the possibility to update the inconsistent database with a repaired one. We provide a language where the user can express any kind of integrity constraint that is possible to be expressed using Answer Set Programming, not limiting ourselves to the integrity constraints of the database management system. Furthermore, we allow the inexperienced user in ASP to express integrity constraints directly in SQL as well.

Our application is built over the PostgreSQL database management system[1]. Because of this particularity, all integrity constraints defined directly in the database satisfy the syntax defined by PostgreSQL.

DRSys implements two methods to compute repairs, which can be selected by the user before the repairing process. Both of them are associated with the minimality issues. Therefore, two distinct minimality measures were introduced, the minimality under cardinality of operations and the minimality under set inclusion.

We also took advantage of the answer set grounder *gringo*, since it allows a direct connection to the database. *Gringo* has a built in scripting language, Lua[2], which allows the interaction with the database directly via the answer set program. Then, instead of developing an application in SQL that would select all tuples from the database and then creating an answer set program where we would add, as facts, all tuples returned from that query, we can do it directly in the logic program.

[1] http://www.postgresql.org/
[2] http://www.lua.org

In this chapter, we describe the interface and the architecture of *DRSys*, emphasizing the features provided by it. Also, we illustrate with several examples, which use the database used in the *TPC-W Benchmark*[3], whose database schema is in Figure 6.1.

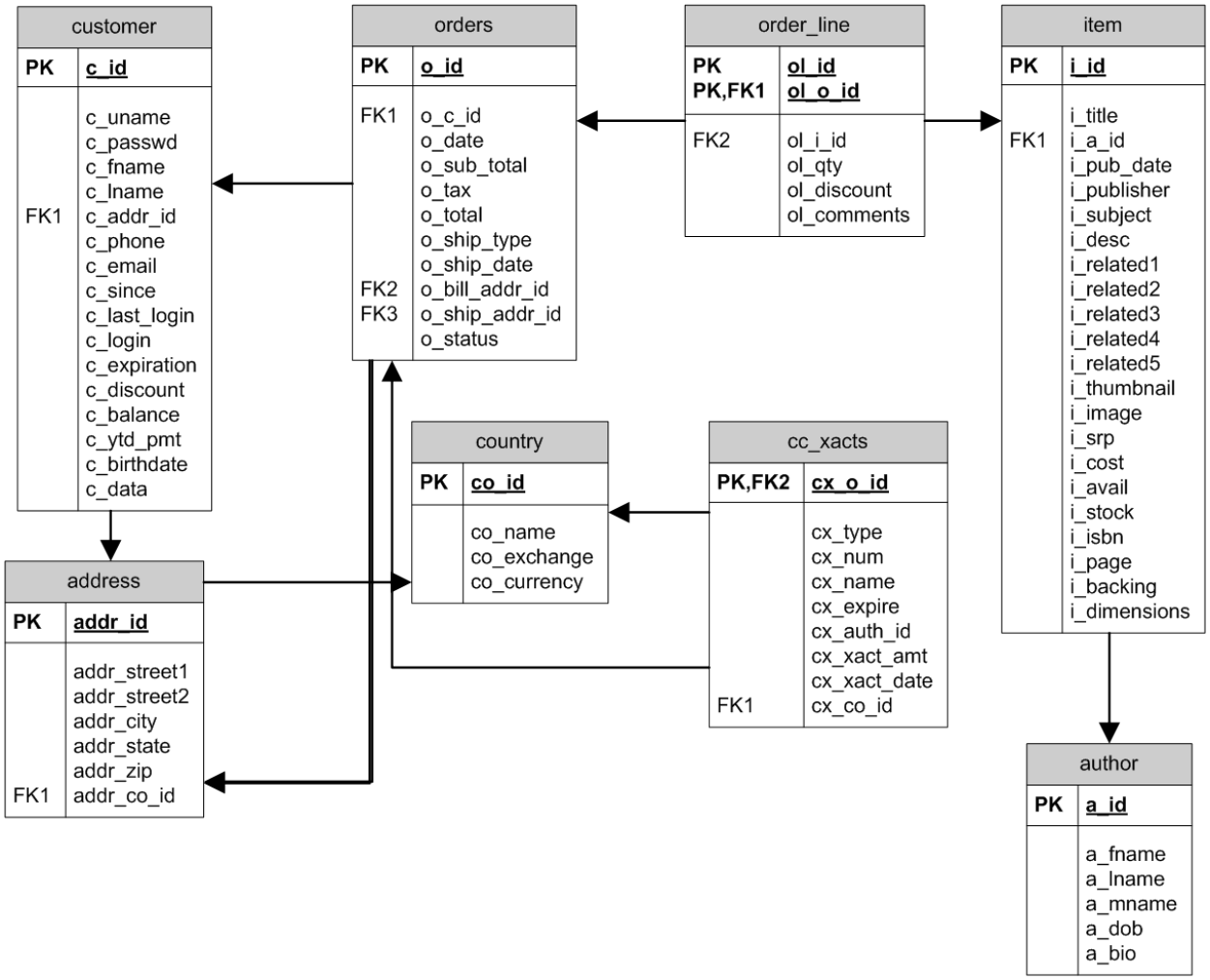

Figure 6.1: *TCP-W* database schema

TPC Benchmark^{TM} *W* (TPC-W) is a transactional web benchmark. We generated the database via a Java script provided by the authors of TPC Benchmark^{TM} W (TPC-W). We extended the database in order to include the integrity constraints described in Figure 6.1. We now proceed with the description of the functionalities an the graphical user interface of *DRSys*.

6.1 Functionalities and Graphical User Interface

In *DRSys* we opted for a *wizard* type of interface, where we have a sequence of menus, each one corresponding to some particular aspect in the repair process. We divide our

[3] http://www.tpc.org/tpcw/

application into the following menus: database connection menu, main menu, integrity constraints edition menu, operations menu, insertion menu, deletion menu, repair menu and the repair choice menu.

Regarding the user's interaction, the repair process goes through these steps: connection to the database, definition of the integrity constraints, edition of the integrity constraints, choice of operations to be performed, insertion parameters, deletion parameters, repairs configurations, repair choice and, finally, database update with the selected repair. We present next a flow chart with all the steps of our application.

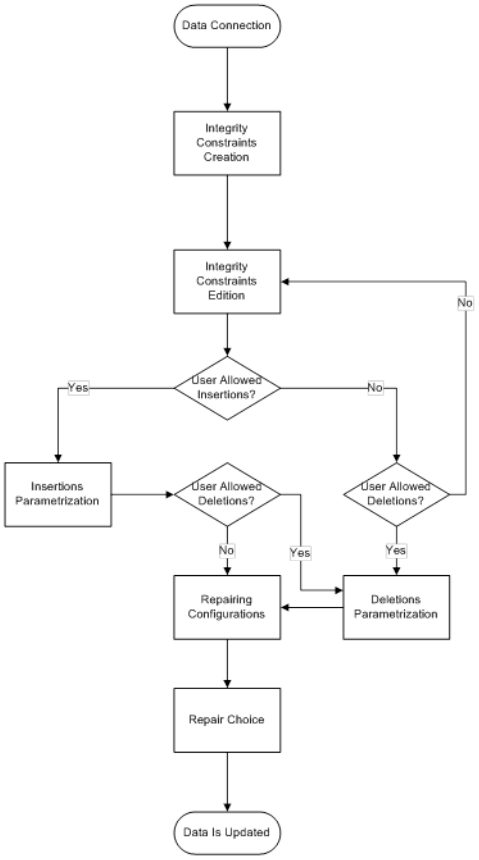

Figure 6.2: *DRSys* flowchart

When the user creates the new integrity constraints, they must be enforced into the database. However, there are still some criteria that are still needed in the application. The user must specify which relations are going to be involved in the process, what kind of operations is he willing to perform (deletions, insertions or both) and, since database repairing is a very complex problem, we also developed some optimizations strategies, automatic ones, and user enforced ones, to ease the repair process. We now describe the interface, by going through the several menus, and discussing the functionalities provided in every one of them.

6.1.1 Database Connection Menu

The first menu in our application is the database connection menu. Here, the user will introduce all the information needed to connect to the desired database - username, password, the name of the database, the IP, and finally, the name of the ODBC connection. To be able to connect to the database directly via the logic program (functionally provided by the answer set grounder), we need to create an ODBC connection, hence being the name of the ODBC connection an input parameter. This menu is depicted in Figure 6.3. After the connection is established, the main menu appears.

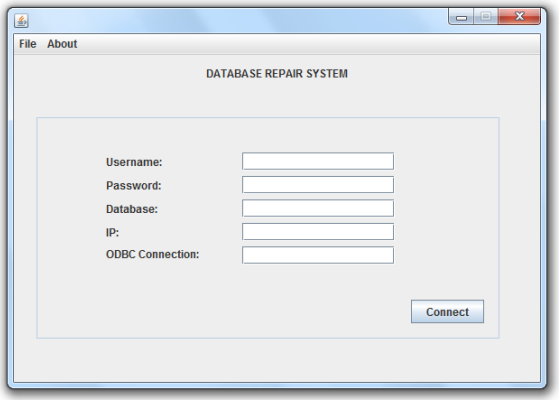

Figure 6.3: Database Connection

6.1.2 Main Menu

Once connected to the database, the main menu shows up. This corresponds to the integrity constraints creation phase. This menu allows the user to define new integrity constraints. In this menu, *DRSys* provides the user with two distinct ways to define integrity constraints: manually and/or automatically. In the manual way, *DRSys* allows the user to express integrity constraints directly in ASP. This way, we provide a very powerful and expressive language to specify integrity constraints. *DRSys* accepts all integrity constraints that can be defined in ASP. Also, since ASP is a very specific domain of computer science and we wanted to make the application as general as possible, without forcing the user to know ASP. Therefore, *DRSys* also provides a way to specify constraints directly in SQL (taking into consideration the syntax of PostgreSQL). For this purpose, *DRSys* provides a converter from SQL to ASP. However, it is very easy to make a mistake while defining a new integrity constraint, whether in ASP or SQL. In order to deal with this problem, *DRSys* offers means to assist in the automatic specification of some integrity constraints. In the automatic way, *DRSys* allows the specification of key constraints, functional dependencies, inclusion dependencies and domain constraints, all of them according to Definitions 2.23,2.24,2.25,2.27 respectively, simply by selecting the attributes that form the constraints and the application constructs the corresponding ASP

code. Figure 6.4 shows the main menu.

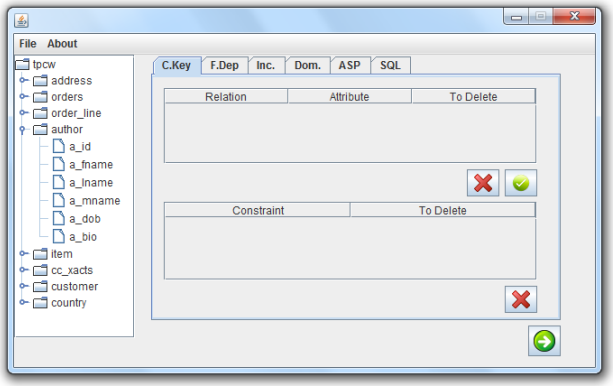

Figure 6.4: Main Menu

The automatically specified integrity constraints are generated with some optimizations. We focus on this part in the next Chapter. Keep in mind that, when defining the integrity constraints automatically, the code generated will be done by the application. However, *DRSys* also allows the user to change this automatically generated code. Therefore, once all the integrity constraints are defined, we proceed to the constraints edition menu.

6.1.3 Constraints Edition Menu

In this menu, we enter the integrity constraints edition phase. *DRSys* allows the user to edit the ASP code generated from all of the previously defined integrity constraints.

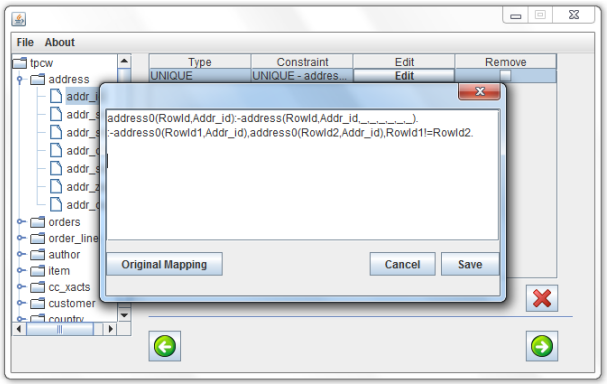

Figure 6.5: Edition Menu

This way, even though some integrity constraints might have been automatically generated by *DRSys*, we allow the user to infer about the generated code, and to change it as well. *DRSys* also allows the user to freely eliminate the integrity constraints defined in the previous menu.

In Figure 6.5 we can see the specification of a key constraint, and the visual tool created to edit the integrity constraint.

After agreeing on the integrity constraints, we proceed to the operations menu.

6.1.4 Operations Menu

In the operations menu, *DRSys* allows the user to choose what kind of operations will be considered during the repair process: if only deletions, if only insertions or both. Figure 6.6 exhibits this menu, together with the provided features. If only insertions were chosen

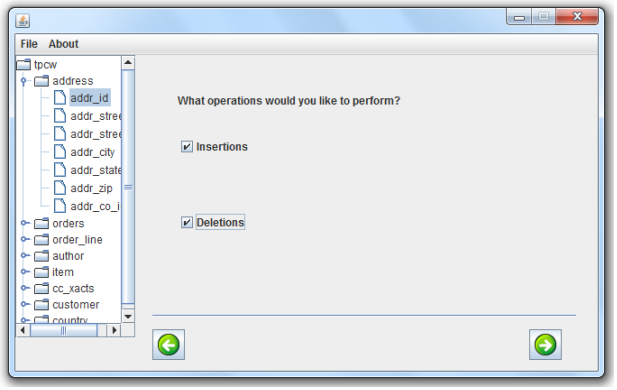

Figure 6.6: Operations Menu

by the user, the next menu to appear will be the insertions menu. If only deletions were chosen by the user, the next menu to appear will be the deletions menu. If both of the operations were chosen, firstly the insertions menu will appear, followed by the deletions menu. Note that, at least one type of operations must be chosen.

6.1.5 Insertions Menu

In this menu, we reach the insertions parametrization phase.

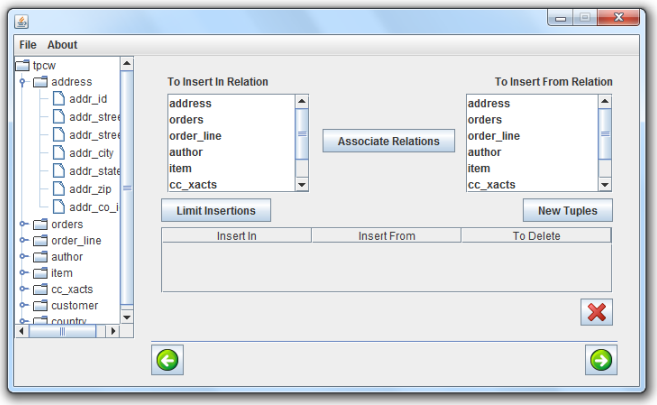

Figure 6.7: Insertions menu

Here, *DRSys* provides the user the possibility to specify the extra source of tuples that are taken into account during the repair process. *DRSys* provides two ways to do this. On the first one, *DRSys* allows the user to select tuples that are stored in some relation of the original database. Figure 6.7 demonstrates this feature.

On the second one, *DRSys* allows the user to manually specify extra tuples. This way, we give more possibilities than the one offered considered only existing tuples in the original database. Figure 6.8 shows the manual specification of new tuples, where the user may manually specify new tuples, that can be inserted in the database.

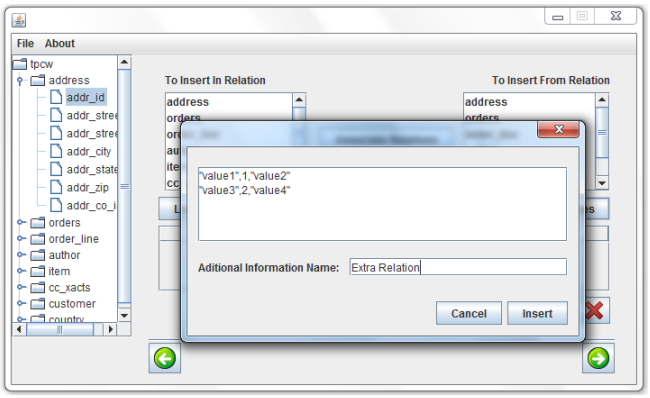

Figure 6.8: Extra Tuples

After deciding the tuples that can be inserted, *DRSys* allows the user to specify where those tuples can be added. For instance, the user may want to insert tuples from *Relation3* in *Relation1* but also tuples from *Relation2* in *Relation4*.

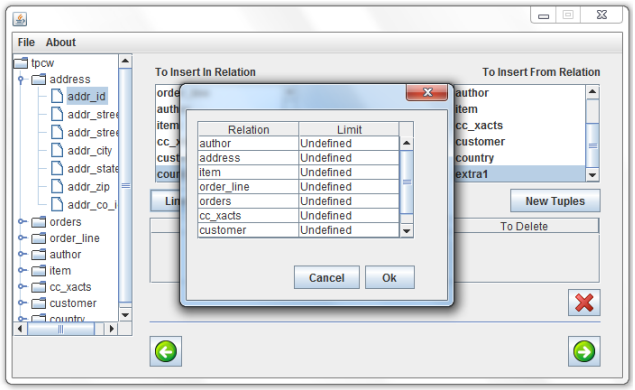

Figure 6.9: Limit Insertions

As a consequence of such association, *DRSys* provides the user one more feature. It allows the user to specify the maximum number of insertions that can be performed in a relation. This way, we can greatly increase performance, as we demonstrate in Chapter 7. Figure 6.9 demonstrates this feature.

6.1.6 Deletions Menu

In this menu, we reach the deletions parametrization phase. Here, very much like the insertion menu, *DRSys* allows the user to select from which relations tuples can be deleted. *DRSys* will only delete tuples from a relation that was selected. As in the insertion menu,

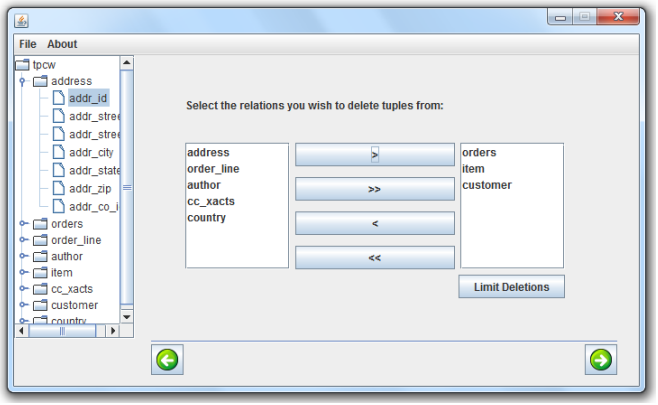

Figure 6.10: Deletions Menu

DRSys also provides the user the possibility to limit the number of deletions that can be performed in a relation. Figure 6.10 illustrates the deletion menu, together with the presented features.

Once the deletion parametrizations are finished, we go to the repair menu.

6.1.7 Repair Menu

In the repair menu, we enter the repairing configuration phase. Figure 6.11 show the repair menu. In this menu, *DRSys* offers the user several features, which we describe next.

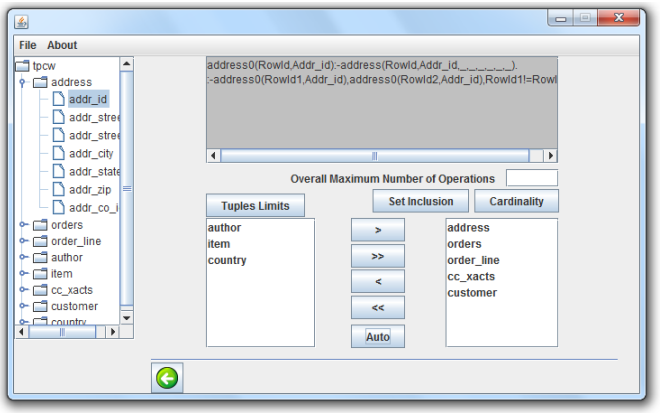

Figure 6.11: Repair Menu

DRSys offers the user, in this menu, the possibility to see all the answer set programming code generated from the defined integrity constraints. This is just for information purposes, not being editable at this point.

DRSys allows the user to define more repairs constraints. It allows the user to mark certain tuples to not be considered for deletion. Although the user may have selected a relation from where tuples may be deleted, the user might not want certain tuples to be deleted. Therefore, in the repair process, such tuples will never be marked as tuples to be deleted. Figure 6.12 show this feature.

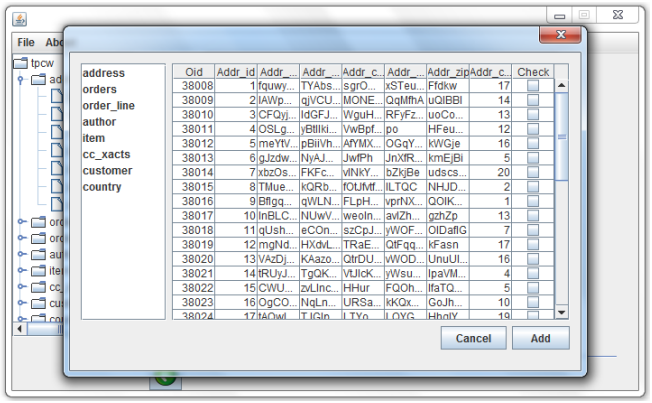

Figure 6.12: Forbid Removals

Furthermore, *DRSys* allows the user to select the relevant relations, i.e., relations that will be taken into consideration during the repair process. *DRSys* allows two distinct ways to select them: an automatic way, by means of a dependency graph, and a manual way. We shall discuss this part in greater detail in the next section.

One other feature that *DRSys* provides in this menu, is the possibility to define the overall maximum number of operations. In the previous menus, the user could limit the maximum number of operations in a specific relation, whereas in here, we limit the overall number of operations.

The greatest feature provided by *DRSys* in this menu, is the possibility to choose which kind of minimality criteria the user wishes to adopt throughout the repair process. Here, *DRSys* allows the user to choose between the set inclusion minimality criteria or the minimality under cardinality of operations criteria. Whichever is chosen, we proceed to the repair choice menu, and consequently, to the repair choice phase.

6.1.8 Repair Choice Menu

According to the minimality criteria chosen, this last menu provides distinct features. We shall begin with the features of minimality under cardinality of operations, and then minimality under set inclusion.

In minimality under cardinality, we allow the user to obtain possible repairs as soon as they are found by clasp. The computation of the repairs is done by a background

process and, whenever a new repair is generated, the *Repair Choice Menu* will show what are the operations that need to be done, while the answer set solver is still running in the background process. Also, and according to the input parameters of clasp, when *DRSys* reaches a repair, the next repair must be better than the previous one, i.e., must perform less operations. This way, *DRSys* keeps generating better and better repairs. *DRSys* also allows the user to stop the process at any time, presenting the user the last repair obtained. Keep in mind that, if the process is stopped, the repair generated may not be minimal, but is still a repair. This is useful when the computation of the minimal repairs takes a lot of time, and the user is already satisfied with the non minimal repair presented. If minimality under cardinality was chosen, we only present the operations that are needed to be performed to obtain consistency with respect to one and only one database repair. Although this is the case, and since there may be several repairs that need to perform the same number of operations, *DRSys* allows the user to ask for all the repairs that perform a specific number of operations.

In minimality under set inclusion, *DRSys* computes and presents all operations that are needed to perform to obtain consistency with respect to all minimal (under set inclusion) repairs only. Although in the answer set part we presented a unsound algorithm, we provide means to retrieve only minimal repairs. We shall go into further details about this topic in the next section as well.

Figure 6.13 shows this last menu.

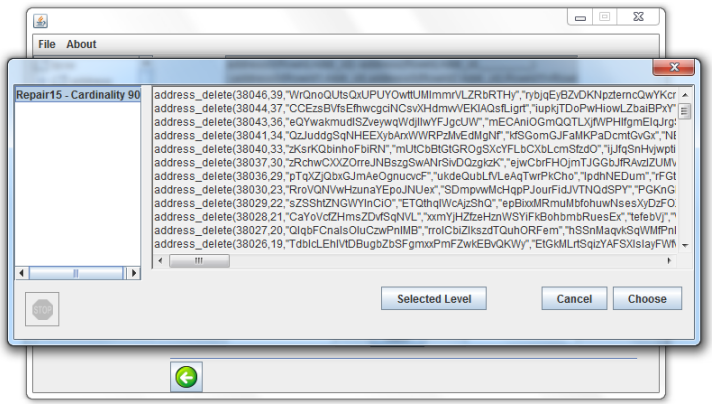

Figure 6.13: Repair Choice

Finally, once the user chooses what repair he will be wanting, *DRSys* updates the database with the newly generated one, leading the database to a consistent state. This corresponds to the final phase, the update phase.

6.2 *DRSys* Architecture

In this section, we present the architecture of *DRSys*, by specifying the behaviour of the application. We also present the optimizations that were adopted in several parts of the

repairing process, in order to speed up the whole process.

The graphical interface of *DRSys* was implemented in Java. To compute the possible repairs, we used, as answer set grounder/solver, *gringo* and *clasp*, respectively. Finally, *DRSys* was built on top of the Database Management System PostgreSQL.

As inputs for the application, we consider the database connection options (username, password, database name, server and the *ODBC* connection), integrity constraints (in answer set programming or SQL), user parametrizations, and operations selection (the ones that will create the repair).

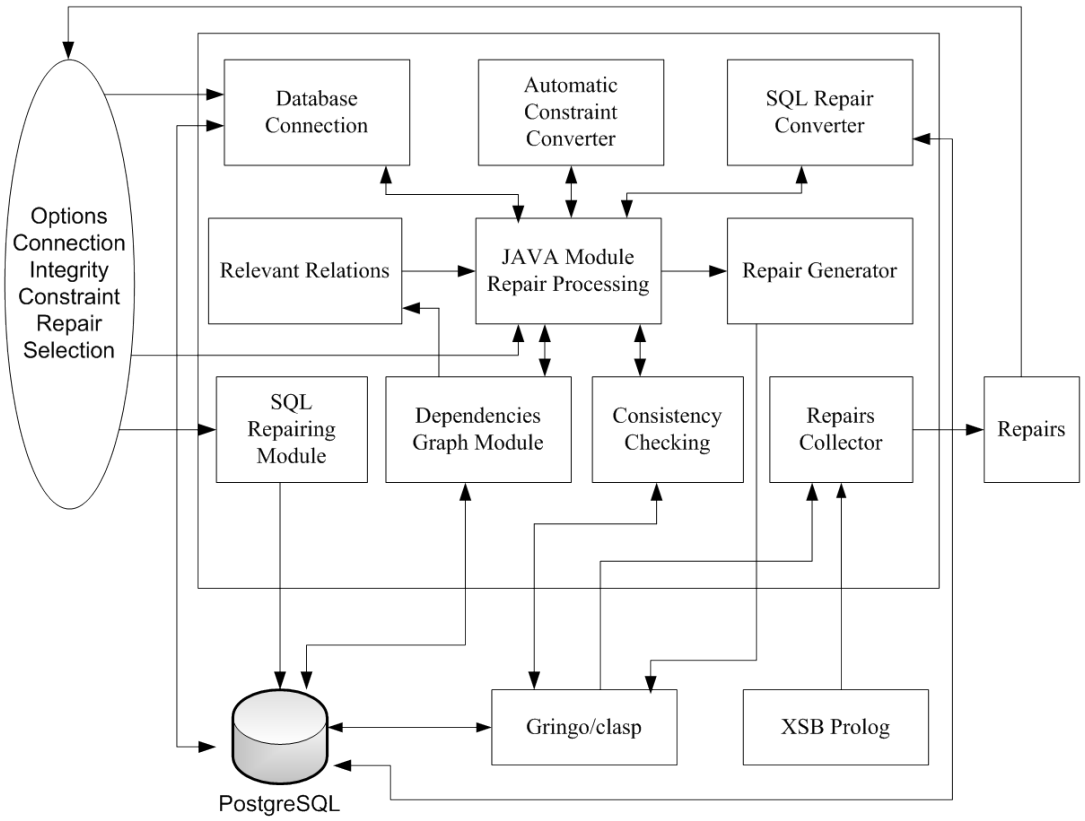

Figure 6.14: *DRSys* Architecture

Figure 6.14 represents the overall design of the architecture. The database connection inputs are received by the *Database Connection Module* that will connect to the specified database. Then, all the interaction will be made with the *JAVA Module*. In it, we developed the graphical user interface (using Java), with which the user interacts. Also, this module represents the connection between all other modules developed. We now specify the whole database repair process, by going through all modules. For that purpose, we shall now describe the architecture of the system, step by step.

The first step is the database connection step. In order to do it, let us describe the module responsible for that, the *Database Connection Module*.

Database Connection Module

Everything in our application that needs to communicative with the database needs to go through the *Database Connection Module*. Besides making the initial connection with the application to the database, this module is also responsible for importing the existing integrity constraints already stored in the database (integrity constraints directly defined in the database management system or user defined constraints in ASP). It is responsible for all queries that need information about the schema as well, for instance, what relations are defined in the database or what are the attributes of a particular relation in the database. Furthermore, it is this module that deletes (inserts) tuples from (into) the database, updating the inconsistent instance with the repaired one, thus restoring consistency. Summarizing, it establishes the connection between the application and the database, and deals with all operations that need access to it.

Let us now go through the repairing process. In the graphical interface, *DRSys* provides means to automatically create integrity constraints of particular classes: keys, functional dependencies, inclusion dependencies, domain constraints, that are directly mapped into Answer Set Programming code. This is the responsibility of the Automatic Constraint Converter Module, which we present next.

Automatic Constraint Converter

By creating the integrity constraints with the automatic mechanism, the user selects the attributes and the relations that are involved in the integrity constraint. Afterwards, the integrity constraints are mapped into an intermediate step, where an internal format is generated, that can be easily read and understood by the user, in order to aid him. Then, according to the internal format, *DRSys* converts the integrity constraints into answer set programming code, ready to be used by the application. We now present the internal format created for the integrity constraints:

- **Keys** - $UNIQUE - relation\_name(att_1, att_2, ..., att_n)$
- **Functional Dependencies** - $F.Dependency - relation\_name(att_{11}, att_{12}, ..., att_{1m})$ $DETERMINES\ relation\_name(att_{21}, att_{22}, ..., att_{2,n})$
- **Inclusion Dependencies** - $Inc.Dependency - relation\_name_1(att_{s1}, att_{s2}, ..., att_{sn})$ $REFERENCES\ relation\_name_2(att_{d1}, att_{d2}, ..., att_{dn})$
- **Domain Constraints** - $DOMAIN - relation\_name\ attribute\_name(dom_1, dom_2, ..., dom_n)$

The meaning of the internal format is very self-explanatory.

As we can see, the internal format of the integrity constraint is made in a way that is completely readable and understandable by the user, requiring only some knowledge about databases.

This module was only created to ease the definition of integrity constraints since, despite not being a hard task to create them in ASP or SQL, it is very easy to make mistakes. This way, we provide a powerful tool to the user, and also, we allow any user, that has some knowledge about databases, to create integrity constraints without needing to know answer set programming.

It may also be the case that the user has some knowledge in SQL, but none in the answer set programming part. Therefore, the user may want to create the integrity constraints by specifying the correspondent SQL code. For this purpose, we developed a module that converts SQL directly into answer set programming code, the *SQL Converter Module*, which we present next.

SQL Repair Converter Module

Since integrity constraints may be specified directly into SQL, we developed a converter from SQL to Answer Set Programming. We allow some specific SQL commands, according to the syntax provided by PostgreSQL, which are:

```
ALTER TABLE tablename ADD PRIMARY KEY (att1,att2,...,attn).
ALTER TABLE tablename ADD CONSTRAINT constraintname PRIMARY KEY
(att1,att2,...,attn).
ALTER TABLE tablename ADD CONSTRAINT constraintname CHECK
(attributename {<,>,<=,>=,!=,=} value).
ALTER TABLE tablename ADD CONSTRAINT constraintname CHECK
(attributename in (value1,value2,...,valuen)).
ALTER TABLE tablename ADD CONSTRAINT constraintname UNIQUE
(att1,att2,...,attn).
ALTER TABLE tablename_1 ADD CONSTRAINT constraintname FOREIGN KEY
(att_1) REFERENCES tablename_2 (att_2).
```

Again, the meaning of each instruction is also self-explanatory, so no explanation will be done[4].

After specifying the constraints, *DRSys* sends the SQL code to the SQL Repair Converter Module.

This module has, as objective, the conversion of SQL code into Answer Set Programming code. It takes as input some SQL statements, and transforms the integrity constraints into answer set programming code, following the transformation pattern presented previously. Notice that, when specifying the integrity constraints directly in SQL, the user loses expressiveness, since he is confined to the integrity constraints provided by PostgreSQL. However, this approach still allows the user to specify integrity constraints that the database management system provides, so it will only lose expressiveness if

[4]For more information, we recommend the user to visit http://www.postgresql.org/docs/9.0/interactive/index.html

more powerful integrity constraints are desired.

Up until now, we have only been talking about the definition of integrity constraints. However, there is more to the database repair than just defining new integrity constraints. *DRSys* needs to know what relations are meant to be involved in the process, in order to generate the logic program. We provide an automatic way to detect which relations are needed in the process, by means of a dependency graph, built by the Dependencies Graph Module.

Dependencies Graph Module

Consider a database with the schema presented in Figure 6.1, and the integrity constraint:

$$\forall_{\kappa_1,\kappa_2,\kappa_3,\kappa_4,\kappa_5,\kappa_6} \neg[order\_line(\kappa_1,\kappa_2,\kappa_3,\kappa_4,\kappa_5,\kappa_6) \wedge \kappa_1 > 1]$$

If no insertions are desired, it is easy to see that only relation *order_line* is necessary to compute the repairs. However, if insertions are desired, we have to take into account many other relations: *orders, customer, address, country, item* and *author*, since there are dependencies between the relations and the integrity constraints.

This module is responsible for building the dependencies graph of this database, i.e., a graph where we represent the connection between relations, with respect to the integrity constraints defined and the operations allowed. The dependencies graph is built and is sent over to the Relevant Relations Module, so that the necessary relations are automatically selected. Let us introduce what the dependencies graph is and how it is built.

The dependencies graph is a directed graph that shows the connections between the relations, with respect to the integrity constraint defined and the operations allowed. The nodes of the graph are name of relations of the database, and the edges correspond to the dependencies between relations.

Let us, once more, consider a database whose schema is depicted in Figure 6.1. Since we allow deletions and insertions as primitives for the repairs, we need to build two dependency graphs: one, to deal with deletions, and the other to deal with insertions. Through the dependencies graph, taking deletion into account, we wish to represent the relations from where we need to deletes tuples. Figure 6.15 shows the dependencies graph associated with the deletions. As we can see, if, for instance, there is a need to delete a tuple from relation *author*, then, following the graph, we need to delete tuples from relation *item* and relation *order_line*. The dependencies graph, considering insertions, is the exact symmetric of the deletions one, with respect to the edges of the graph. In Figure 6.16 we show this other graph. Therefore, if, in the repair process, there is the need to insert a tuple in, for instance, relation *order_line*, tuples in *item* and *author* must be inserted as well, so that the integrity constraints are still satisfied.

The resulting graphs are then sent to the relevant relations module, that will select the

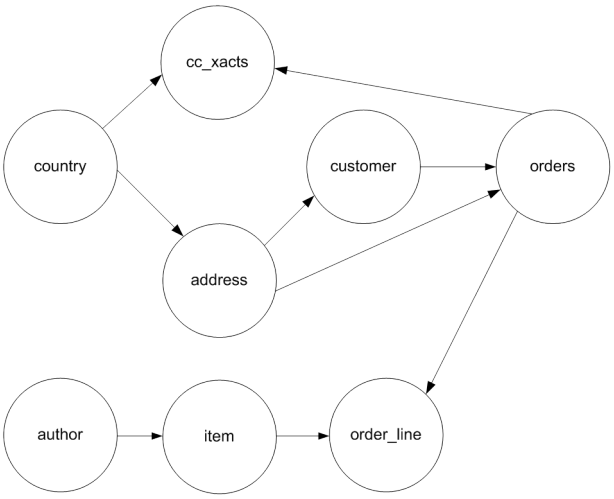

Figure 6.15: Dependencies Graph considering deletions

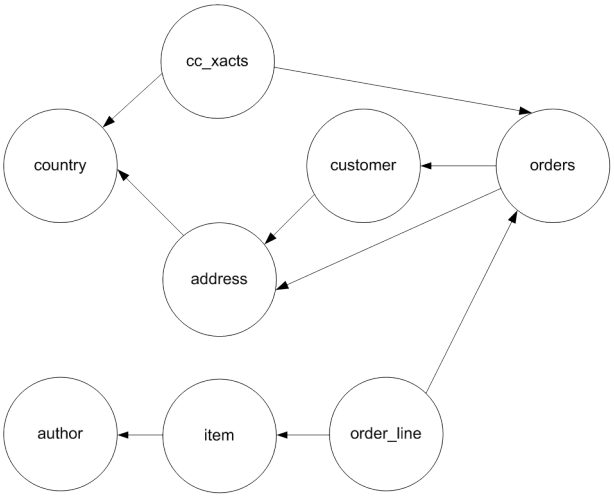

Figure 6.16: Dependencies Graph considering insertions

relations that will be involved in the repair process, taken into consideration the newly created integrity constraints.

Relevant Relations Module

In this module, we determine the relevant relations that must be involved in the repair process. But first, let us clarify what a relevant relation is. Given a set of integrity constraints IC (the new defined integrity constraints) and an operation $op \in \{deletion, insertion\}$, and two dependencies graphs (the deletion one and the insertion one), a relevant relation is a relation that:

- is a relation that is directly involved in any integrity constraint of IC;
- is a relation that, in the dependencies graph according to op, depends on a relevant

relation.

More intuitively, a relevant relation is a relation that is directly expressed in a integrity constraint, or a relation that depends on that relation, directly or indirectly (via some other relations), according to the type of operations allowed.

For instance, consider the following integrity constraint, regarding the database schema of Figure 6.1:

$$\forall_{x_0,x_1,...,x_6} \neg[author(x_0, x_1, ..., x_6), x_2 =' Richard']$$

The initial relevant relation is the relation *author*, since it is directly involved in the integrity constraint. However, if we consider deletions in the repair process, relations *item* and *order_line* are also considered relevant relations. If, on the other hand, only insertions were taken into account, only relation *author* was considered a relevant relation.

However, the algorithm we developed to build the dependencies graph is very conservative, i.e., it may consider as relevant relations, relations that, in fact, are not. Suppose that, in the database, we wanted to enforce the following integrity constraint:

$$\forall_{x_0,x_1,...,x_6} \neg[author(x_0, x_1, ..., x_6), x_1 > 1]$$

The previous integrity constraint states that the value for the attribute x_1 in every tuple of relation *author*, cannot be bigger than 1. Now consider that such integrity constraint leads the database to an inconsistent state. Therefore, a repair is needed. Now consider that we only want to delete tuples in relation *author* and *item*. Then, according to our algorithm and the dependency graph, *DRSys* would consider, as relevant relations, the following relations: *author*, *item* and *order_line*, when, in fact, only the relations *author* and *item* should be considered, since, if we do not allow deletions in relation *item*, there is no need to select relation *order_line* as a relevant relation. For this particular reason, and since relevant relations must be selected, instead of providing only an automatic mechanism to select the relevant relations, we also allow the user to select them manually, since it may be better in some cases to manually select the relevant relations.

As input of this module, we receive the dependencies graph, computed in the previous module. Also, the newly defined integrity constraints must be passed as input to this module. Then, we analyse the relations involved in those constraints. After we know the direct involved relations, for each of those relations, and considering the type of operations allowed, we go through the dependencies graphs and compute the additional relevant relations. Once we obtain the relevant relations, we can proceed to the repair process.

In order to greatly increase performance, we divide the repair process in two distinct phases: the consistency checking phase (*Consistency Checking Module*), where we verify if the database becomes inconsistent when trying to enforce the new integrity constraints in the database and, if so, we proceed to the repair phase (*Repair Generator Module*), where we generate the possible repairs. We introduce both modules next.

Consistency Module Checking

This module is responsible for checking consistency in the database, with respect to the new integrity constraints defined by the user. Here, we create a logic program, very similar to the one presented in the transformation function 5.1. There are although some slight differences. In this new logic program, we are only concerned with what exists in the main database, since it is only there that we check the (in)consistency. That way, we don't have rules of the type (5.2). Also, we are not interested here in computing the repairs, so, we don't need to know if any atom will be kept, deleted or inserted. Therefore, rules of the type (5.3) and (5.4) are excluded from this repair program. All the remaining rules will be present, with one more difference. We replace the predicate p_keep_N by the predicate p_N. Finally, we add to the logic program the integrity constraints defined by the user and the logic program is evaluated. If no stable models of the logic program are generated, inconsistency is present in the database. It is easy to accept this, since, if no stable models were generated, some integrity constraint present in the logic program was trigerred, stating that some integrity constraint is being violated in the database. On the other hand, if any stable model is generated, consistency is achieved and there is no need to compute the repairs.

This module is simply an optimization module. It was indeed possible to verify consistency by running only the repair program. If the database did not become inconsistent with respect to the new integrity constraints, we just had to wait until the empty model was generated (considering the tuples to be deleted or inserted), meaning that no tuple was meant to be deleted nor inserted. It is also easy to see that, if no atom is meant to be deleted nor inserted, it is because there is no inconsistency. Therefore, the empty model is generated. However, before the generation of the stable models, a lot of other models would be generated. In the sequel, computational time would be wasted. For this purpose, we developed the *Consistency Module Checking*.

If inconsistency is detected by the Consistency Checking Module, then we proceed to the repair phase (*Repair Generator Module*).

Repair Generator Module

This module is responsible for computing the repairs, with respect to a database repair problem P. According to the minimality chosen by the user, a logic program of the form $\varphi^C(P)$ or $\varphi^S(P)$ is created. However, according to the minimality criteria chosen, the behaviour of the Repair Generator Module changes. If minimality under cardinality is chosen, a logic program of the form $\varphi^C(P)$ is created an ran in *gringo / clasp*. The stable models generated correspond to possible repairs, existing a one-to-one correspondence of the answer sets with the repairs. However, if minimality under set inclusion is chosen, *DRSys* works in a different way. Still, a logic program of the form $\varphi^S(P)$ is created. However, as we have showed, this approach is not sound. So, we went a little bit further. Analysing the answer sets generated, we build a new logic program, a *Prolog* program.

There, we build a list of lists, such that the number of lists is equal to the number of answer sets generated. Also, given an answer set X, the created list will contain all the atoms belonging to $\Delta(X)$. Then, in the *Prolog* program, we generate the minimal repairs under set inclusion. The answers generated by XSB will represent the operations that need to be performed in order to achieve consistency in the database. After computing the answers, they are sent back to the *DRSys* application, and *DRSys* shows the user the possible ways to restore consistency, by showing the user the operations that need to be performed. We went a little bit outside of the scope of this dissertation, by using some other technologies, but, we provide a sound and complete approach this way.

Here, however, we do not create the repair program straight forward as shown before. We introduce some optimizations statements, to enhance the performance of the application. We present here some modifications to the logic program created.

Optimizations

The logic programs $\varphi^C(P)$ and $\varphi^S(P)$ are both created, by having as base $\varphi(P)$. Therefore, all modifications that we present, are made in the latter, so that both the previous ones inherit as well these optimizations. Take into account that the modifications rely on the language provided by the answer set solver, so we shall take full advantage of that. Some optimizations are done automatically, and others are done by the user, manually. Note that if the user decides to use some knowledge to increase the performance of *DRSys*, if the parametrization of some features is not done well, it is possible that no repairs are generated.

- **Importing Integrity Constraints** - When the relevant relations are defined, *DRSys* queries the database to obtain the already defined constraints that are related to those relations and maps them into the logic program according to the transformation function presented. It only obtains the integrity constraints associated with the relevant relations, instead of simply adding all integrity constraints present in the database.

- **Deletions/Insertions Optimization** - The generation of possible tuples to be deleted, presented in (5.3), will slightly change. Instead of those rules, we create the following generation rules for every relevant relation N[5]:

$$0\{delete(x_0) : p_N(\bar{x})\}n.$$
$$p\_keep_N(\bar{x}) \leftarrow p_N(\bar{x}), not\ delete(x_0).$$
$$delete_N(x_0) \leftarrow p_N(\bar{x}), delete(x_0).$$

  We first generate all possible atoms to be deleted, and the atoms to be kept are the ones that are not meant to be deleted. It is better to generate the deletions first

[5]Recall that x_0 corresponds to the *Rowid* of a tuple, being it also the first term of $\bar{x}$

instead of the whole repaired instance since, usually, the number of deletions is much smaller than the atoms to be kept. Also, while generating the atoms that are meant to be deleted, we are also giving a maximum number of deletions, which is expressed by n. This n is an optimization parameter introduced by the user, which stands for the maximum number of deletions that can be done in relation N. Note that the last rule will allow us to know from where we need to delete the tuple, which will be important when updating the database.

Analogously, a similar approach is taken when dealing with the atoms that are meant to be inserted, based on (5.4). For every relation N from where we can obtain the tuples to insert, we add the following rules:

$$0\{insert(x_0) : p\_keep^i_N(\bar{x})\}n.$$
$$p\_keep_N(\bar{x}) \leftarrow p\_keep^i_N(\bar{x}), insert(x_0).$$
$$insert_N(\bar{x}) \leftarrow p\_keep^i_N(\bar{x}), insert(x_0).$$

- **Repair Constraints** - We allow the user to have some influence regarding possible tuples to be deleted and the number of tuples to be deleted. For that purpose, we created two distinct optimizations.

  - **Forbidding deletions in a relation** - In *DRSys*, we allow the user to specify that there cannot be any deletion in a particular relation. In order to explicitly say that that, the set of rules presented in (5.3) are modified to:

    $$p\_keep_N(\bar{x}) \leftarrow p_N(\bar{x}).$$

    where N is the relation where we are forbid to delete atoms, and the deletions/insertions optimizations are ignored.

    With this rule, we force the atoms of relation N of the main database to be present in the repaired version, meaning that no tuple can be deleted from relation N.

  - **Forbidding deletions of specific tuples** - In *DRSys*, we allow the user to manually specify tuples that cannot be deleted. In order to forbid certain tuples to be deleted, we add integrity constraints of the following type to the repair program:

    $$\bot \leftarrow delete(x_0), p_N(\bar{x}).$$

    where N is the relation where the tuple belongs to.

    By adding this integrity constraint, there cannot be a model where an atom that is in the main database is meant to be deleted, regarding the specific tuples chosen by the user.

- **Maximum Overall Number of Operations** - Besides providing limits on the number of deletions and insertions in a particular relation, in *DRSys* we allow the specification of a maximum overall number of operations. For instance, the user may say that there cannot be more than 10 deletions in *order_line* and no more than 20 deletions in *item*, but, besides that, the user may also say that here cannot be more than 5 operations globally. In order to represent this in the logic program, instead of introducing the rules:

$$count(M) \leftarrow M = \#sum[delete(DRowid), insert(IRowid)].$$
$$\#minimize[count(M) = M].$$

we introduce an additional rule, that will limit the maximum number of operations:

$$count(M) \leftarrow M = \#sum[delete(DRowid), insert(IRowid)].$$
$$\bot \leftarrow count(M), M > n.$$
$$\#minimize[count(M) = M].$$

where n is the overall maximum number of operations. Notice that this optimization can only be done considering minimality under cardinality of operations.

- **Projection of attributes** - Finally, we made one last change in the transformation function. Instead of considering all attributes, while specifying an integrity constraints, we only consider the relevant attributes. By relevant attributes, we mean the ones that are directly related with the integrity constraint. In order to do this, we make the projection of the relevant attributes, and then map the integrity constraint regarding only those attributes. Consider the following example:

Example 2. *Consider a database with the schema depicted in Figure 6.1. Suppose we wish to add a check constraint in relation country of the form* $CC(country, co\_exchange, >, 1000)$. *Following the mapping we introduced, we would create the following rule:*

$$\bot \leftarrow p\_keep_{country}(Att_0, Att_1, Att_2, Att_3, Att_4), Att_3 > 1000.$$

Now, instead of doing that mapping, we create the following rules:

$$p\_keep_{CC(country,co\_exchange,>,1000)}(Att) \leftarrow p\_keep_{country}(\_, \_, \_, Att, \_).$$
$$\bot \leftarrow p\_keep_{CC(country,co\_exchange,>,1000)}(Att), Att > 1000.$$

This optimization applies to all the classes of integrity constraints presented in the transformation function. Since the mapping, considering this optimization, is very similar to the one in the previous example, we will not go into further details. Also, this mapping is used in the automatic constraint converter module and in the SQL repair converter module.

This new mapping is very effective and is much better than the straight forward mapping presented before. Although we have more rules in the logic program, the grounded program will be significantly reduced. This way, performance is greatly increased, being the overall time needed to perform the repairs reduced.

The output of this module will be the models generated from the logic program created, which correspond to the possible repairs or, the solutions provided by XSB, which correspond to the operations needed to perform in order to restore consistency, according to the minimal repairs under set inclusion.

Once the repairs are generated, we must extract, from the stable models, the operations that need to be done, that is, the information regarding the tuples that are meant to be deleted and the tuples that are meant to be inserted. For this purpose, we developed the *Repairs Collector Module*, which we present next.

Repair Collector

This module has, as input, the stable models generated from the evaluation of the logic program previously created. The answer generated by XSB will not be treated here, since they are already treated directly in XSB. Then, this module isolates the atoms that are meant to be deleted and the atoms that are meant to be inserted. Since we have atoms of the form $delete_N(c_0)$ and $insert_N(c_0)$ in the answer sets, we can know exactly from which relation we need to delete tuples and into which relation we need to insert tuples. Let us define two extracting functions, a deletion extracting function, to collect the atoms that are meant to be deleted, and an insertion extracting function, to collect the atoms that are meant to be inserted.

Definition 6.1 (Deletion Extracting Function)**.** *Let P be a database repair problem and $\varphi(P)^C$ be the corresponding logic program. Assume X is an answer set of $\varphi(P)^C$. Then, let deletion extracting function $\Delta^{d\prime}$ be defined as follows:*

$$\begin{aligned} \Delta^{d\prime}_N(X) &= \{to\_delete_N(c_0) | delete_N(c_0) \in X\} \\ \Delta^{d\prime}(X) &= \bigcup_{N \in \mathcal{N}} \Delta^{d\prime}_N(X) \end{aligned}$$

Definition 6.2 (Insertion Extracting Function)**.** *Let P be a database repair problem and $\varphi(P)^C$ be the corresponding logic program. Assume X is an answer set ofr $\varphi(P)^C$. Then, let insertion extracting function $\Delta^{i\prime}$ be defined as follows:*

$$\begin{aligned} \Delta^{i\prime}_N(X) &= \{to\_insert_N(\bar{c}) | insert_N(\bar{c}) \in X\} \\ \Delta^{i\prime}(X) &= \bigcup_{N \in \mathcal{N}} \Delta^{i\prime}_N(X) \end{aligned}$$

For each stable model X in the set of stable models received as input, the set of atoms

that are meant to be deleted and inserted form the output of this module.

Since it is possible to exist several minimal repairs, we separate all the modifications from distinct repairs into distinct groups, and send them to the user, so that he can see which operations he wants to perform to obtain a repair. Once the user chooses the operations he wants, the corresponding SQL code needs to be generated, in order to update the inconsistent database, generating a repaired version, and that is what is done in the *SQL Repairing Module*.

SQL Repairing Module

This module receives as input the atoms that are meant to be deleted, in the form of $\Delta^{d\prime}(X)$ and the atoms that are meant to be inserted, in the form of $\Delta^{i\prime}(X)$. Then, the corresponding SQL needs to be generated. We perform the following operations:

- For each atom $to\_delete_N(oid) \in \Delta^{d\prime}(X)$, we create the following SQL statement:

  ```
  delete from N where oid = oid
  ```

  The *oid* attribute represents the object identifier, which is, in PostgreSQL, very similar to the concept of *Rowid*, since it allows a tuple to be uniquely identified in the whole database.

- For each atom $to\_insert_N(\bar{c}) \in \Delta^{i\prime}(X)$, we create the following SQL statement:

  ```
  insert into N values(c1,c2,...,cn)
  ```

By performing these operations, we restore consistency to the database. To finalize the repair process, *DRSys* also stores the integrity constraints defined by the user in the database, in ASP format.

We have shown the life cycle of the database repair. As we could see, to avoid computing the repairs straight forward, we developed some optimization mechanisms that greatly reduce the computation time needed. We next present the experimental study done in *DRSys*, where we take some important conclusions about its behaviour.

7

Experimental Evaluation

In this chapter, we present the experimental tests regarding the performance of *DRSys*. We investigate the influence of the optimization parameters, the influence of the number of integrity constraints and also the influence of the size of the database.

The experiments were performed on a Intel® Core™ i7 CPU 920 @ 2.67 GHz, with 6.00 GB RAM, with Windows 7 Professional , 64 bit operating-system. The database instances were stored on PostgreSQL 8.4 – PostgreSQL Global Development Group. All the answer set programs were run using, as the answer set grounder, Gringo, version 3.0.3, as the answer set solver, and clasp version 1.3.4.

For every experiment[1], we used the database of the *TPC Benchmark™ W*, whose database schema is presented in Figure 6.1. For some particular tests, we introduced new relations to the database, which will be mentioned when appropriate. Also, all of the experiments were realized three times each. The results presented are the average of such results.

7.1 Experimental Results

We investigated several properties in our application. These properties are:

- Influence of the number of irrelevant relations involved in the repair process;
- Influence of the number of integrity constraints involved in the repair process;
- Influence of the number of operations per relation and overall number of operations in the repair process;

[1]The database instances used for all the experiments are available online, and can be downloaded at http://sourceforge.net/projects/drsys/files/drsys1.0/.

- Influence of the number of irrelevant integrity constraints in the repair process;
- Influence of the size of the database in the repair process.

For each of the tests done, we created a specific scenario, so that the influence of the specific factor could be understood.

7.1.1 Influence of the Number of Irrelevant Relations Involved in the Repair Process

For this experiment, we wanted to test if, by adding some relations that are totally irrelevant considering the newly created integrity constraints, the total time necessary to compute the repairs would be affected. For this purpose, we fixed a database with 10.000 tuples and used the following integrity constraints:

$$F_1 = \forall_{x_0,x_1,x_2,x_3,x_4} \neg[country(x_0, x_1, x_2, x_3, x_4) \wedge x_1 > 1]$$
$$F_2 = \forall_{x_0,x_1,x_2,x_3,x_4,x_5,x_6} \neg[author(x_0, x_1, x_2, x_3, x_4, x_5, x_6) \wedge x_1 > 1]$$
$$F_3 = \forall_{x_0,x_1,...,x_{11}} \exists_{y_0,y_1,...,y_6} [\neg orders(x_0, x_1, ..., x_{11}) \vee (author(y_0, y_1, ..., y_6) \wedge x_3 = y_1)]$$

The integrity constraint F_1 states that there cannot be a tuple in relation *country*, such that the value of the attribute *co_id* is bigger than 1. The integrity constraint F_2 states that there cannot be a tuple in relation *author*, such that the value of the attribute *a_id* is bigger than 1. The integrity constraint F_3 states that values for the attribute *o_date* of relation *orders* must exist as the values of the attribute *a_id* of relation *author*.

The relevant relations are the ones determined by the algorithm. Considering F_1, they are: *country*, *address*, *customer*, *orders*, *order_line* and *cc_xacts*. Considering F_2, the relevant relations are: *author*, *item*, *order_line*. If we consider both of them together, all relations are relevant relations. Considering F_3, the relevant relations are: *author*, *item*, *order_line*, *orders* and *cc_xacts*. Also, the added relations do not have any integrity constraints associated with them.

All of the irrelevant relations used here contained 1.000 tuples, and all of them had arity 4. Also, we only considered deletions, and computed minimal repairs under cardinality of operations. Furthermore, we considered that *DRSys* could delete tuples from every relevant relation.

We realized four experiments. In the first one, we used the integrity constraint F_1. Then, we tested the time needed using only the relevant relations, then, with one irrelevant relation with 1.000 tuples, then two irrelevant relations (each with 1.000 tuples), and so on, until eight irrelevant relations (each with 1.000 tuples). We did the same using only F_2 and then using F_1 together with F_2. In the fourth experiment, we used the integrity constraint F_3 and did the same as in the previous experiments. The results of this experiment can be seen in Figure 7.1.

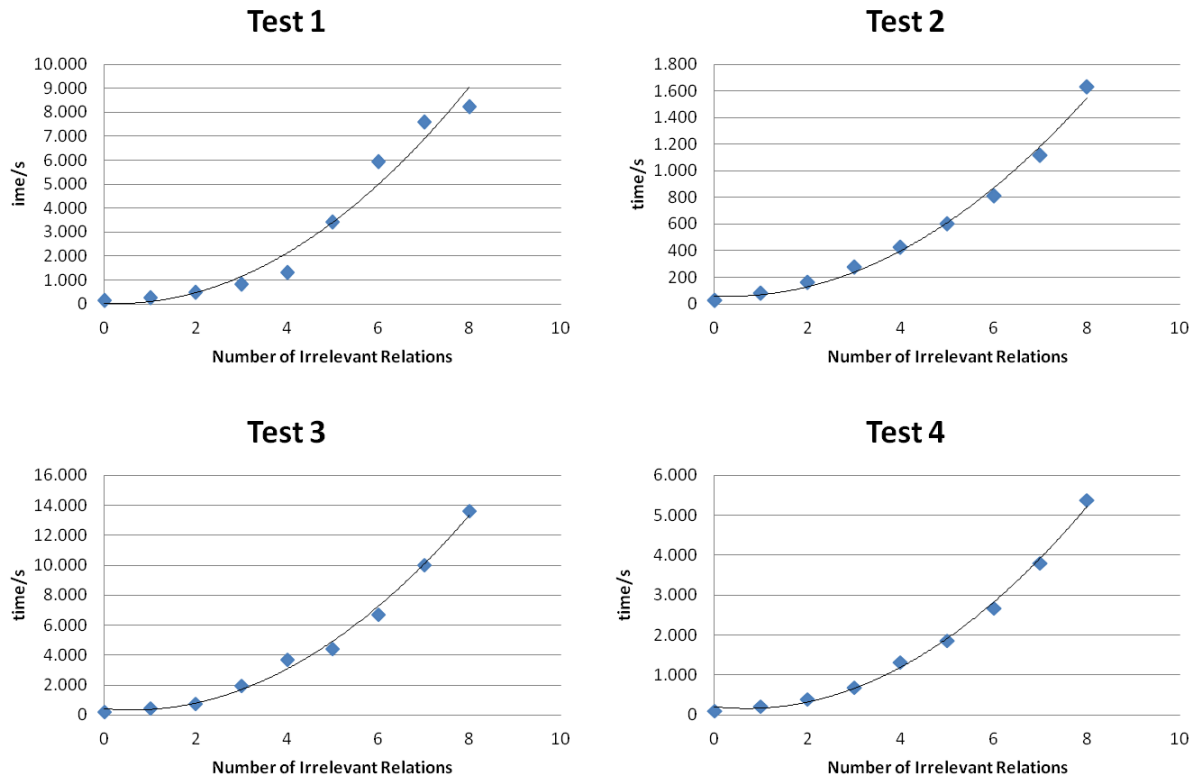

Figure 7.1: Influence of the number of relations involved in the repair process

As we can see from the results, adding extra relations to the repair process is severely prejudicial. It is highly recommended that, whenever the user does not have information about the database schema, to automatically choose the relevant relations. The algorithm to automatically determine the relevant relations may be conservative, but still, the user is able to introduce even more irrelevant relations than the algorithm. Otherwise, irrelevant relations can be inserted in the process being the repair process severally affected.

By adding irrelevant relations, we are adding more rules of the type:

$$0\{delete(x_0) : p\_keep_N(\bar{x})\}n$$
$$p\_keep(\bar{x}) \leftarrow p\_keep_N(\bar{x}), not\ delete(x_0).$$

By doing so, the ground program grows drastically, and not in a linear way (as expected). Being that the case, the search space is significantly bigger, taking much longer to reach the desired minimal repairs.

With these results, we can also state that the type of integrity constraints also affects the performance of *DRSys*, since we altered the integrity constraints used, and the number of direct relations involved in the integrity constraints as well.

We also wanted to know if the performance would be affected if, instead of adding

several irrelevant tuples from a set of relations, we added only one relation with more irrelevant tuples. For instance, instead of adding 8.000 tuples from eight different relations (each of them with 1.000 tuples), we wanted to know what would happen if we added 8.000 tuples but from one single relation. The results did no suffer significant changes, as expected, taken into account the implementation of *DRSys*, so we do not present any graphical results for this experiment.

7.1.2 Influence of the Number of Integrity Constraints Involved in the Repair Process

For this experiment, we wanted to test how the addition of new integrity constraints to the database affects the overall time necessary to compute the repairs. We considered a database with 5.000 tuples and the following integrity constraints:

$$F_1 = \forall_{\kappa_0,\kappa_1,...\kappa_4,\varepsilon_0,\varepsilon_1,...,\varepsilon_6} \neg[country(\kappa_0,\kappa_1,...,\kappa_4) \wedge author(\varepsilon_0,\varepsilon_1,...,\varepsilon_6) \wedge \kappa_1 = \varepsilon_1]$$
$$F_2 = \forall_{\kappa_0,\kappa_1,...,\kappa_7} \neg[address(\kappa_0,\kappa_1,\kappa_2,\kappa_3,\kappa_4,\kappa_5,\kappa_6,\kappa_7) \wedge \kappa_7 > 25]$$
$$F_3 = \forall_{\kappa_0,\kappa_1,...,\kappa_4} \neg[country(\kappa_0,\kappa_1,\kappa_2,\kappa_3,\kappa_4) \wedge \kappa_1 > 40]$$
$$F_4 = \forall_{\kappa_0,\kappa_1,...,\kappa_6} \neg[address(\kappa_0,\kappa_1,\kappa_2,\kappa_3,\kappa_4,\kappa_5,\kappa_6) \wedge \kappa_1 < 45]$$

The integrity constraints F_1 forbids tuples (from relation *country* and from relation *author*) to have the same value for the attributes *co_id* and a_id respectively. The integrity constraint F_2 forbids tuples from relation *address* to have values for the attribute $addr\_co\_id$ bigger than 25. Integrity constraint F_3 forbids tuples from relation *country* to have a value for the attribute co_id bigger than 40 and, finally, integrity constraint F_4 forbids tuples from relation *address* to have a value for the attribute a_id smaller than 45.

The relevant relations considered for these experiments were all the relations in the database.

For all the experiments, we only considered deletions, and computed minimal repairs under cardinality of operations. Also, we considered that is was possible to delete tuples from all the relevant relations.

We performed three different experiments, where we simply increased the number of integrity constraints to be enforced in the database. In the first experiment, we started with, besides the integrity constraints already defined in the database that are related to the relevant relations, the integrity constraint F_1. Then, added F_2, then F_3 and finally, F_4. Then, in the second experiment, we started with, besides the integrity constraints already defined in the database, the integrity constraint F_4. Then, added F_3, then F_2 and finally, F_1. Finally, in the third experiment, we started with, besides the integrity constraints already defined in the database, the integrity constraint F_3. Then, added F_2, then F_4 and finally, F_1. Figure 7.2 shows the results of these experiments.

*This particular test, at the end of 30.000 seconds, still did no have reached a minimal repair. We opted to terminate the test without letting it reach a minimal repair.

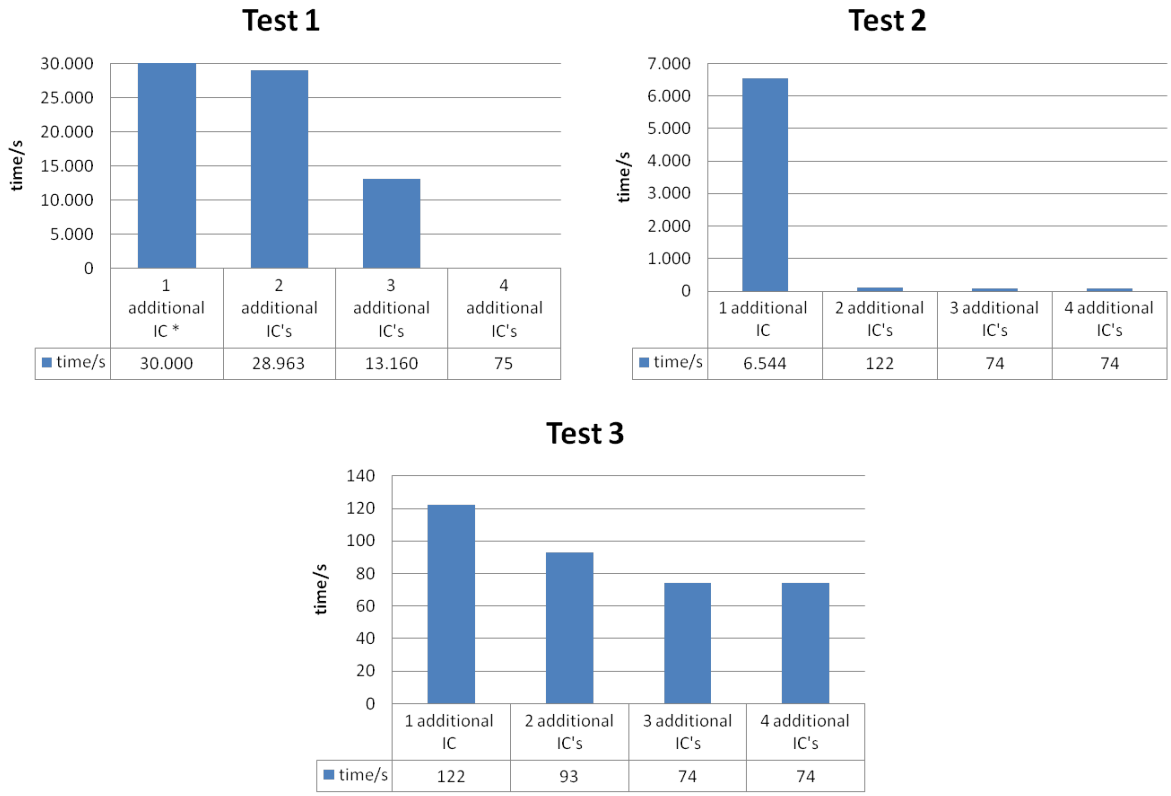

Figure 7.2: Influence of the number of constraints involved in the repair process

As we can observe from the results, the kind of integrity constraint greatly affects the overall performance of *DRSys*. We could start to justify these results by comparing the size of the ground program. However, it is not significant enough to explain such leaps. We need to investigate the way *clasp* works. The primary algorithm of *clasp* relies on conflict-driven nogood learning, a technique that proved very successful for satisfiability checking [GKNS07]. Conflict analysis is the procedure that finds the reason for a conflict an tries to resolve it. It tells the solver that there exists no solution for the problem in a certain search space, and indicates a new search space to continue the search [ZMMM01]. A nogood is a set of literals, expressing a constraint violated by any assignment containing those literals. Furthermore, *clasp* is embedded with several heuristics that allow a faster computation of the problem [DGKS10]. We can speculate that, as we keep introducing more integrity constraints, we are generating more *nogoods* and, therefore, more conflicts and, because of this, *clasp* is able to prune the search space more efficiently, i.e., it discards some solutions, reducing the remaining search space, and, therefore, increasing performance.

When we changed the order in which we added the integrity constraints, we were also changing the pruning quality of *clasp*. In the second test, we may infer that the pruning was more effective than the first one, and, in the third test, the pruning was even

more effective.

7.1.3 Influence of the number of operations per relation and overall number of operations in the repair process

For this experiment, we wanted to test how the variation of the maximum number of operations in a relation, by fixing an integrity constraint, affected the overall time necessary to compute the repairs. We also wanted to test how the variation of the overall maximum number of operations, by fixing an integrity constraints, affected the overall time necessary to compute the repairs. Furthermore, we wanted to compare both approaches with each other, whenever operations were only realized in one, and only one relation. For this purpose, we considered a database with 20.000 tuples and the following integrity constraint:

$$\forall_{x_0,...,x_6,y_0,...,y_6} \neg[order\_line(x_0,...,x_6) \wedge order\_line(y_0,...,y_6) \wedge x_0 \neq y_0 \wedge x_4 = y_4 \wedge x_5 < y_5]$$

With the previous integrity constraint, we forbid two distinct tuples from relation *order_line* to have the same values for the attributes ol_qty, with one value for the attribute $ol\_discount$ of a tuple being smaller than the other value for the same attribute for the other tuple. Furthermore, we considered the following maximum number of tuples to be deleted in relation *order_line*: 255 tuples (which corresponds to the minimal repair), 300 tuples, 400 tuples, 500 tuples and unbounded (the computation of the repair straight forward without optimizations). We did the same considering these values as the number of overall number of operations permitted.

For this experiment, we only considered deletions, and computed minimal repairs under cardinality of operations. As relevant relations, we considered those generated by our algorithm, being them only the relation *order_line*. We considered that is was only possible to delete tuples from that relation as well. When limiting the overall maximum number of repairs, we stopped the process when *DRSys* computed the first minimal repair (in case there should be more than one minimal repair).

The results of these experiments are both shown in Figure 7.3.

As we can observe, if a maximum number of operations in a relation, or an overall maximum number of operations is given as input, the computational time required to generate the repairs is greatly reduced, thus improving the performance of *DRSys*. Although the ground program is not altered, we speculate that, since we are restricting the maximum number of operations in a relation (or the overall maximum number of operations), we are, once more, generating more *nogoods* and more conflicts, and, therefore, *clasp* prunes the search space, reducing the necessary time to reach the optimum repair. Notice that the direct limitation of the number of operations in a relation is slightly better than the overall limitation. However, since both options increase performance, if operations must be realized in more than one relation, the use of both criteria is possible, increasing always performance of *DRSys*.

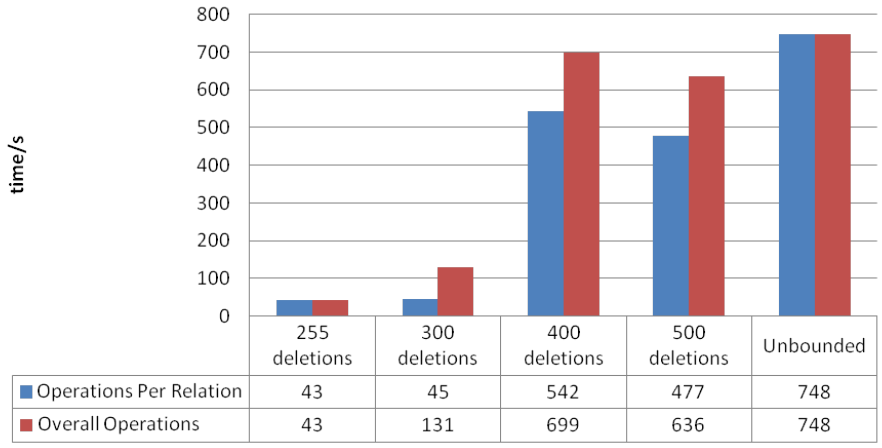

Figure 7.3: Influence of the number of user defined deletions in the repair process

Notice that, if operations can be realized in more than one relation, limiting the maximum number of operations in a relation can lead to the generation of repairs that are not minimal. Consider the following relation depicted in Table 7.1, the source of extra tuples in Table 7.2 and the following integrity constraint:

$$\forall_{AId,CId}\exists_{Name}.[\neg accounts(AId, CId) \vee customers(CId, Name)]$$

customers

CustomerId	Name
111	John
222	Peter
333	Anna

accounts

AccountId	CustomerId
1	111
2	222
3	333
4	444
5	444
6	444
7	444
8	555
9	666
10	666

Table 7.1: Inconsistent Database

As we can see, this database is clearly violating the integrity constraint. If we consider deletions and insertions, and minimality under cardinality of operations, the minimal repair will be the one where the tuples $\langle 444, Michael\rangle$ and $\langle 555, Susan\rangle$ and $\langle 666, Richard\rangle$ are added to the *Customers* relation. However, imagine that the user said that only one insertion could be performed in relation *Customers*. This way, the repair generated would be the one where the tuples $\langle 8, 555\rangle$, $\langle 9, 666\rangle$, and $\langle 10, 666\rangle$ would be deleted and the tuple $\langle 444, Michael\rangle$ would be inserted. This would not be minimal but, with the information

Extra

CustomerId	Name
444	Michael
555	Susan
666	Richard

Table 7.2: Extra tuples

that the user provided, it is the best repair possible.

By looking at the results once more, we can see that, the less we limit the number of operations, the worst the time gets. Strangely, the time needed to get to the minimal repairs when we limited the number of operations to 400 is greater than when we limited the number of operations to 500 (still however smaller than the straight forward approach). Considering the implementation of *DRSys*, there is no reason to explain this results. Perhaps it is a matter of the implementation of *clasp* itself, but, nevertheless, the result is better than the unbounded approach.

7.1.4 Influence of the Number of Irrelevant Integrity Constraints

For this experiment, we wanted to test how the presence of irrelevant integrity constraints affected the overall time needed to compute the repairs. Let us illustrate this problem with an example: consider the relations depicted in Table 7.3 and the following integrity constraint:

account	
Acid	Cid
11	11
22	11
33	22
44	44

client	
Cid	Name
11	Richard
22	John

Table 7.3: *Account* and *Client* relations

$$F_1 = \forall_{Cid,Acid} \exists_{Name} [\neg account(Acid, Cid) \vee client(Cid, Name)]$$

The previous integrity constraint simply states that there cannot be an account associated with an client that is not present in the database. Furthermore, consider that the following integrity constraints were already defined and store in the database:

$$F_2 = \forall_{Acid,Cid_1,Cid_2} \neg [account(Acid, Cid_1) \wedge account(Acid, Cid_2) \wedge Cid_1 \neq Cid_2]$$
$$F_3 = \forall_{Cid,Name_1,Name_2} \neg [client(Cid, Name_1) \wedge client(Cid, Name_2) \wedge Name_1 \neq Name_2]$$

The previous integrity constraints state that a client is identified by its Id and that an account is identified by an account Id as well. It is clear that there is a violation the integrity constraint F_1. Consider as well that we are only dealing with deletions. It is clear that none of the integrity constraints F_2 and F_3 will never be violated, since we are not dealing with insertions. Therefore, we wanted to see if, by not introducing such integrity constraints in the repair program, the overall time to compute the repairs would be affected. Therefore, we realized two distinct experiments. For this purpose, we considered some integrity constraints already used in the second test:

$$F_1 = \forall_{\kappa_0,\kappa_1,\ldots,\kappa_4} \neg[country(\kappa_0,\kappa_1,\kappa_2,\kappa_3,\kappa_4) \wedge \kappa_1 > 40]$$
$$F_2 = \forall_{\kappa_0,\kappa_1,\ldots,\kappa_6} \neg[address(\kappa_0,\kappa_1,\kappa_2,\kappa_3,\kappa_4,\kappa_5,\kappa_6) \wedge \kappa_1 < 45]$$

In each of the experiments, we considered only the relevant relations generated by our algorithm. According to F_1, the relevant relations are: *country*, *address*, *customer*, *orders*, *order_line* and *cc_xacts*. In the second one, they are the same, except for the *country* relation. In both experiments, we considered only deletions, and minimality under cardinality of operations. Also, we considered that *DRSys* could delete tuples from all of the previous relations.

In the first experiment, we first generated repairs repairs using all integrity constraints that were imported by *DRSys* together with F_1. Then, we ran the exact same experiment, excluding all key constraints, since they would never be violated in this case. In the second experiment, we did the same, but using the integrity constraint F_2, instead of F_1. In both experiments, the time needed to reach the optimal solution when considering the key constraints or not was almost the same. There is a slight difference, but, comparing to the overall time necessary, it is irrelevant. The results are shown in Figure 7.4.

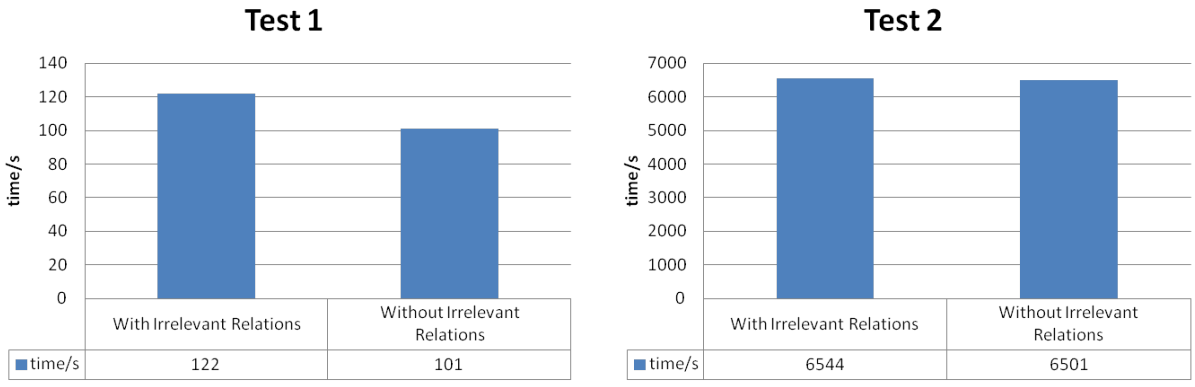

Figure 7.4: Influence of the number irrelevant integrity constraints in the repair process

In order to justify this results, we should, once more, explore the implementation of *clasp*. Although the primary algorithm of *clasp* relies on constraint-driven nogood learning, it also incorporates various advanced Boolean constraint solving techniques,

being one of them early conflict detection [DGKS10]. This way, it may be the case that, since *clasp* detects the conflicts early, it may be able to detect that some other integrity constraints will never be violated. This way, although the resulting ground program may be bigger when including the irrelevant integrity constraints, we speculate that *clasp* is able to ignore those integrity constraints, since they are not violated.

7.1.5 Influence of the of Size of the Database in the Repair Process

For this experiment, we wanted to test how changing the size of the database, by fixing an integrity constraint, affects the time to generate the repairs. For this purpose, we considered the same integrity constraints as in the first test.

$$F_1 = \forall_{x_0,x_1,x_2,x_3,x_4} \neg[country(x_0, x_1, x_2, x_3, x_4) \wedge x_1 > 1]$$
$$F_2 = \forall_{x_0,x_1,x_2,x_3,x_4,x_5,x_6} \neg[author(x_0, x_1, x_2, x_3, x_4, x_5, x_6) \wedge x_1 > 1]$$
$$F_3 = \forall_{x_0,x_1,...,x_{11}} \exists_{y_0,y_1,...,y_6} [\neg orders(x_0, x_1, ..., x_{11}) \vee (author(y_0, y_1, ..., y_6) \wedge x_3 = y_1)]$$

Once more, the relevant relations are the ones determined by our algorithm. Considering F_1, they are: *country*, *address*, *customer*, *orders*, *order_line* and *cc_xacts*. Considering F_2, the relevant relations are: *author*, *item*, *order_line*. Considering F_3, the relevant relations are: *author*, *item*, *order_line*, *orders*, and *cc_xacts*. If we consider both of them together, all relations are relevant relations. Also, the added relations do not have any integrity constraints associated with them.

For the experiments, we only considered deletions, and computed minimal repairs under cardinality of operations. For F_1 and F_2 and F_3, we considered that we could delete tuples from every relevant relation.

We realized four different experiments. In the first one, we used the integrity constraint F_1. In the second one, we used the integrity constraint F_2. In the third one, we used both integrity constraints. In the fourth one, we only used the integrity constraint F_4. In every one of the experiments, we changed the size of the database. We started with one database with 1.000 tuples, then 2.000 tuples, then 3.000 tuples and so on, until we reached 15.000 tuples. The results of this experiment can be seen in Figure 7.5. As we can see, if the size of the database grows, the necessary time to compute the repairs grows exponentially. If we have more tuples in the database, more facts will be added to the logic program. As a result of such additions, the size of the ground program grows significantly, increasing the search space as well, resulting in the exponential growth of the necessary time. This shows that the direct approach to solve the inconsistency is very heavy when the number of tuples in a database is elevated as well. It is greatly advised that the user inputs some information that may be used to optimize the computation, enhancing the performance of *DRSys*.

We also performed some tests regarding whether it would take more time to perform

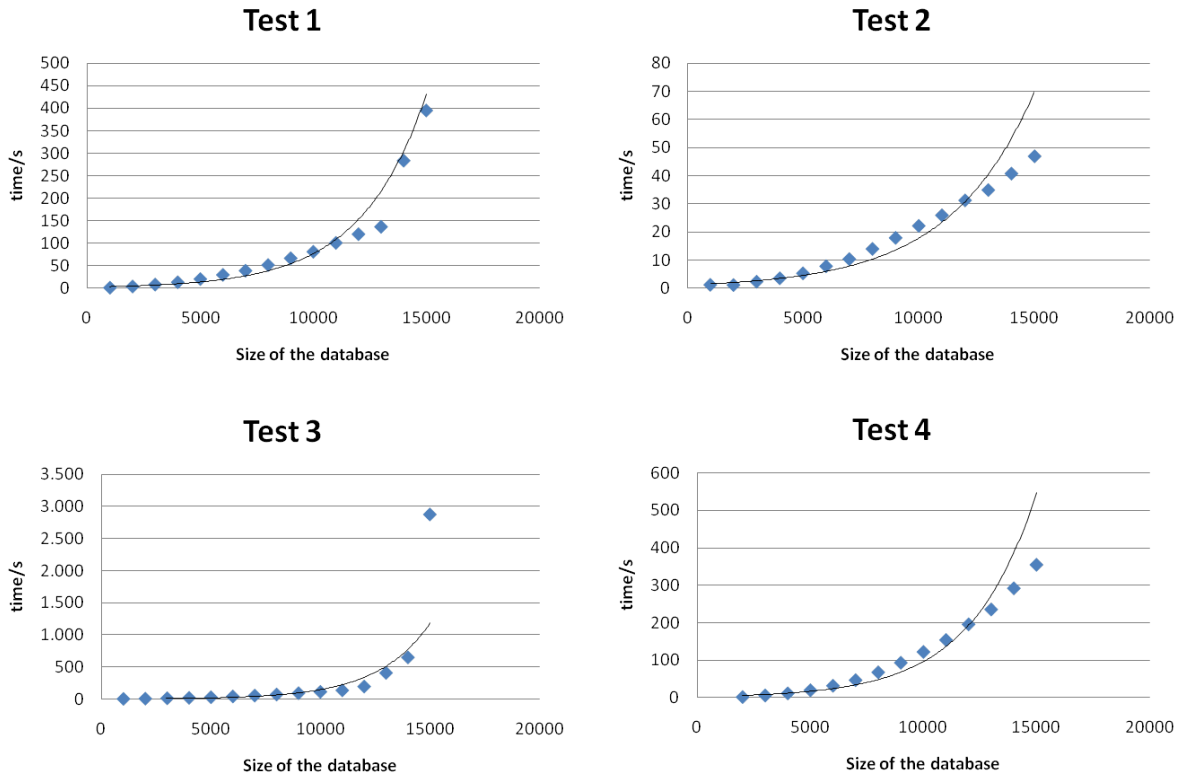

Figure 7.5: Influence of the size of the database in the repair process

repairs by deleting tuples or by inserting tuples. According to such tests, the time needed to reach a repair where, for example, 10.000 are meant to be deleted is approximately the same as the time needed to reach a repair where 10.000 are meant to be inserted. Therefore, we do not present any graphical result.

7.2 Comparison

DRSys is an application developed to repair databases. Besides generating all possible repairs, it allows the user to actually update the database, replacing the old inconsistent instance by a new consistent instance (with respect to the integrity constraints defined).

In this section, we compare our results with the results of other works done in the database repair area, taken into account the works presented in Chapter 4. We shall focus on the following aspects: functionalities, quality, applicability, integrity constraints mapping, performance and parametrization requirements.

7.2.1 Functionalities

The majority of the work done in the area of database repairing was the theoretical formulation of algorithms to repair databases[ABC99, ABC00, ABC03, VS09, KL09, Wij05,

Bry97, Cho07, CM05, RD00, Wij03, SMG10, GGZ03]. No real applications have been developed in these cases. We, on the other hand, besides formalizing the algorithm, created an actual application that allows the user to repair the database. Also, almost every work in the database repairing literature only considers minimality under set inclusion. In *DRSys*, besides providing two minimality criteria, we allow the user to freely choose between them. Also, we formally proved the soundness and completeness of our transformation function, and, informally, we argued for the soundness and completeness including the minimality criteria as well.

In *DRSys*, we perform repairs by inserting or deleting tuples, giving the user the option to choose whichever possibility he desires. In the insertion part, we do not allow the insertion of the $NULL$ value, so *DRSys* offers the user the possibility to introduce an extra source of tuples that will be taken into account in the repair process. In the database repairing literature, some works presented a third option, the option to also perform updates[Wij03, Wij05, CZ06, FPL+01], replacing a value of an attribute by a $NULL$ value, by another constant, or simply by a variable. However, by introducing variables in the database, we are simply introducing unknown values, which may not be desired. In the census data repair[FPL+01], the update makes sense, however, the domain of the attributes was very small, and, therefore, special rules to deal with the problem could be added into a logic program, to specify how to update a certain value. Suppose that, in a database, we found an inconsistent tuple from a relation *person*, which has, as attributes, the following: *Id, Name, Sex*. Also, the value that was inconsistent was the value for the attribute *Sex*. Since the domain of that attribute is only $\{Male, Female\}$, it is easy to express in a rule that the sex of a person should be either female or male. Now suppose another example, on the same relation, but the value that is inconsistent is the value of the attribute Id. In Portugal, we have around 10.000.000 habitants. Suppose that the domain of the attribute *Id* is then the set of natural numbers from 1 to 10.000.000. It is not feasible to express a rule on how to update this value.

In *DRSys* we also allow the user to have some influence in the repair process, by introducing repair constraints, i.e., constraints that can limit the number of maximum deletions, limit the number of maximum insertions, limit the number of maximum overall operations, and limit the tuples to be deleted. These were based on some ideas introduced in [GGZ01], where the authors introduced rules that could influence the final outcome.

Furthermore, we provide an automatic way of generating some integrity constraints directly into ASP format, which, as to our knowledge, has not been done in any of past works. We provide the automatic creation of key constraints, functional dependencies, inclusion dependencies and domain constraints, providing the user a graphical display to do it, in a convenient and intuitive manner. This way, we allow the user without any knowledge regarding ASP to use *DRSys*.

We also allow the user to select the relevant relations to be involved in the repair process, in an automatic or in a manual way. In [Can07], the authors also provide an

automatic way to select relevant relations, also through the use of a dependency graph.

As we could see, *DRSys* is a very complete application, that takes into consideration a lot of features that can help the user and the application itself, by enhancing its performance.

7.2.2 Quality

In *DRSys*, we traded performance for quality. As we have shown, we present a general, sound and complete approach to deal with database repairing. This way, we present an algorithm that can deal with any kind of integrity constraint that can be expressed through ASP, providing a powerful and expressive language to the user. Other approaches, such as [KL09, BIG10, CM11], could only deal with functional dependencies, making the repair problem more restrict. Others, such as [RD00, LLL00, LLL01, BSIBD09, BSI+10], traded quality for performance, i.e., developed algorithms that are, on the one hand, faster than the ones we presented, but, on the other hand, are not complete. *DRSys* provides a general, sound and complete approach. It may not be as fast as others approaches, but it is more general.

7.2.3 Applicability

DRSys is an application built to work with centralized databases, where the user has permissions the change the data. Database repairing is a technique that is mostly used under these circumstances, and *DRSys* follows it. In all of the work done in the database repairing literature, as far as we know, all studies and applications developed considering database repairing, are only done in a centralized environment. There are actually some studies considering databases that are stored in distinct sources [VS09, CZ06], but, in those cases, algorithms for merging databases were developed, resulting in a unique centralized database, and then, it was over this database that database repairing was used. In these cases, although it may seem that database repairing is being done over distributed databases, in fact, it is being done over centralized databases. Our approach is no different from those. It was built to work in a centralized environment.

7.2.4 Integrity Constraints Mapping into Logic Programs

Many of the studies done only considered a small fraction of integrity constraints. In [KL09, BIG10, CM11], the authors only considered functional dependencies. In [BSI+10, LLL00], the authors only took into consideration duplicate tuple detection. In [GL97], denial constraints were not covered. In many other studies [GGZ03, ABC00, ABC03, CB00, ABC99, BB03], inclusion dependencies were not covered. In [KL09, BIG10, CM11], only functional dependencies were covered. Furthermore, many integrity constraints can be expressed as a universal constraint, i.e., a universally quantified first order formula. Keys can be expressed this way, functional dependencies can be expressed this way and denial constraints as well. Then, some studies, where answer set programming

was used, provided transformation functions based only on this most general integrity constraint[BB03, ABC03, ABC00, MB07, GGZ03]. In *DRSys*, we consider all of those integrity constraints as separately. This way, we could create a more specific mapping for such integrity constraints, increasing performance, without losing generality, since all of them are covered, simply separately. Also, in those studies disjunctive logic programs were used, increasing the problem complexity to a higher class. In *DRSys*, we focused on logic programs without disjunction.

We also consider that integrity constraints are properties that allow us to have more consistent information. We consider them as being extremely important, and not subject to changes, as opposed to [CM11], since otherwise, there would be no point in developing such a concept.

7.2.5 Performance and Scalability

In terms of performance and scalability, there have been several studies that allow the computation of repairs to be faster than the one we presented. However, the are some subtleties that need to be further analysed. In some studies, [KL09, BIG10, CM11], since the authors only focused on a particular class of integrity constraints (functional dependencies), specialized algorithms were developed in order to increase performance, losing, this way, generality.

There have also been some other approaches that are still faster than *DRSys*. In [RD00, LLL00, LLL01, BSIBD09, BSI+10], database repairs were computed based on greedy algorithms. These algorithms deal with heuristics. Therefore, they are not complete, and, the repair obtained may not be a satisfactory repair, depending on the parameters established for the algorithm. However, in terms of performance, they are very powerful. In this case, quality was traded for performance. These algorithms can also scale better than ours.

In *DRSys*, we traded performance for generality. Because of that, and since and given the complexity of the problem we address, we introduced some user parametrization to increase performance. These parametrizations can have great influence in the repair process and still generate the optimal repair, allowing the system to scale better.

7.2.6 Parametrization Requirements

Some of the work done heavily relied on the use of algorithms that required the parametrization of certain variables by the user [RD00, LLL00, LLL01, BSIBD09, BSI+10]. For instance, in greedy algorithms, a good heuristic is desired, otherwise bad repairs can be made. Also, in some data cleaning algorithms [BSI+10, LLL01, BSIBD09], parameters for the duplicate detection algorithms are needed, in order compare two distinct tuples and know if they are indeed duplicates. This also affects the quality of the repair, since we are introducing more input parameters, besides the ones used in the repair algorithms themselves. According to those algorithms, the outcome of the applications designed may

suffer several changes, having worse or better repairs, according to those parameters. Also, in those cases, there is a high trade-off. If the parameters are very good and accurate, the repairs will also be good, but the computational time will greatly increase. On the other hand, if worse parameters are chosen, worse repairs are generated, however, the computational time needed is a lot inferior.

Our approach does not rely on any user parametrization to obtain correct results. We offer a free parametrization transformation function. We do allow the user to input some parametrization, in order to enhance the performance of *DRSys*, but, if none is given, *DRSys* reaches minimal repairs (although taking more time). This way, there is no need for automatic learning of the parameters values, or user experience calibration of them, being our application independent in this way.

The greatest achievement of this dissertation was the creation of a real application that allows the specification of new integrity constraints at any point in time. Although in the database repairing literature there are many studies regarding database repairing, they are just theoretical studies, not existing real applications. In *DRSys*, we provide the user with a declarative language, expressive enough to define powerful integrity constraints at any point in time, that are not supported by the database management systems, because they are too complex and heavy to maintain. *DRSys* is a working application, providing means to repair databases, as well as some additional features that can greatly improve performance, and compute repairs under two distinct minimality criteria.

8

Conclusions and Future Work

Throughout this dissertation, we presented *DRSys*, an application to repair databases using Answer Set Programming. We provide the user means to define new integrity constraints and enforce them into the database. If the database becomes inconsistent when doing so, our application restores consistency to the database, by deleting and/or adding tuples from/to the database. We presented a transformation function (the mapping of the database repair problem into Answer Set Programming language) and proved its soundness and completeness. We also presented our approach taking minimality of the repairs into consideration, under set inclusion or under cardinality of operations. We introduced some tweaks to enhance the process of repairs, by introducing some optimizations statements to greatly reduce the number of candidate repairs. We described the interface and the architecture of *DRSys*. Finally, we showed some experimental results, regarding the performance of our application in different scenarios, in order to demonstrate its behaviour towards several features. We have shown that, although *DRSys* does not easily scale, there are some optimizations that can be made to ease the process and allow greater performance. We must also take into consideration that the process of repairing a database is not meant to be done in a daily basis, but very rarely. Therefore, if the process of repairing databases is, on one hand, a very complex and slow process, on the other hand the number of times that it must be used throughout the life time of the database is reduced. However, there are still improvements that can be made in order to increase performance, there are some different approaches to this problem that can circumvent some main issues. We present some ideas of future work next.

The use of *NULL* values

In our approach, in order to deal with insertions, and without the use of $NULL$ values, the user has to manually specify a source of extra tuples. However, it would be a good idea if repairs were done completely automatically, without any kind of user dependence. To this purpose, the value $NULL$ should be allowed in the domain of every attribute. In the literature, the is no consent on the meaning of the $NULL$ value. There is no actual global accepted semantics [Rei82, Mai83]. Also, how do we extend the relational model to accept this special value? Studies have been made in this area. In [BB06], the authors introduce an approach to deal with this problem, and the work done in [MB07] already contemplates this problem, following the semantics introduced in [BB06]. Furthermore, if the value $NULL$ is a possible value of the domains, a 3-valued logic would be interesting to use, instead of the 2-valued logic used here. Therefore, an extensive study on the Well-Founded Semantics should be made, in order to try to incorporate this situation. Consider the following relation and the following integrity constraint:

$$\forall_{Cid,Acid}\exists_{Name}[\neg account(Acid, Cid) \vee client(Cid, Name)]$$

account	
Acid	Cid
11	11
22	11
33	22
44	44

client	
Cid	Name
11	Richard
22	John

Table 8.1: Inclusion Dependency - Null Values

In our approach, without specifying the source of extra tuples, the only possible minimal repair would be to delete the tuple $account(44, 44)$. However, considering the $NULL$ value, one possible repair would be to insert the tuple $client(44, NULL)$. This way, consistency would also be restored.

Updates versus Deletions and Insertions

In *DRSys*, only deletions and/or insertions are allowed. With deletions, it may be the case that we are deleting a lot of correct information. Imagine a relation with 20 attributes, and there is a tuple in that relation where only one value of one attribute is inconsistent, with respect to some integrity constraint, and the others are right. If we delete this tuple, we are deleting good information, since one value is wrong and the rest is correct. Also, it may be that, in some cases, deleting information is not at all desired, even though inconsistency is present in the database. A good example of a real life situation where this is

true, is the census situation, presented in [FPL+01], which used updates as a repair primitive. However, this was only possible in that case by introducing some rules that, in a way, "showed" how to update the values. Also, the domain of each attribute was very reduced. In the general cases, one approach could be to manually specify a domain, which would represent the constants with which the values of attributes could be replaced with. However, this would, once again, make database repair process less automatic, since it was up to the user to define that domain. Another possibility may be to update the erroneous values to the $NULL$ value. Despite existing studies on this area, there has not been an approach that convinced us of the efficiency of updates. Further study on this area is still required, nevertheless, it is still a very interesting approach.

Incremental Database Repairing

In *DRSys*, and in almost every study made in the database repair problem that uses logic programming to represent the database repair problem, the first step is to generate all possible modifications, considering all tuples from relevant relations of the database. However, it would be interesting if an automatic procedure was developed that performed the repairs in the opposite way as we have been doing so far. We would start with a repair that would perform 1 operation. If no models were generated, we would then try again the process by performing $1 + \alpha$ operations. If no models were generated, we would consider $1 + 2\alpha$ operations, and so on, until possible repairs were generated. Since the number of inconsistent tuples is generally way less than the number of tuples in the database, this would probably increase the performance of the application. However, the α parameter should be well defined. Also, with this approach, we may not get the optimal repair, but we get very close repairs, substantially increasing the performance of the application. That could be done in *DRSys*, by simply performing several iterations of the algorithm, always restricting the overall maximum number of operations. However, if using *DRSys*, in every iteration we would need to ground the logic program created. If the database has a big amount of tuples, the grounding will take some time as well. A good way to address this problem could be the use of *iclingo*[1] [1], which is an incremental ASP system implemented on top of *clingo*. It is based on the idea that the grounder, as well as the solver are implemented, in a stateful way. Thus, both keep their previous states while increasing an incremental parameter. Therefore, since iclingo is implemented in a stateful way, it "remembers" the grounding program. When we perform the first iteration, the ground program is created. When performing a new iteration, the previous grounding is "remembered", therefore, only the new part of the programs needs to be grounded, and so on, until one repair is found.

[11] For more information about iclingo, we recommend the reader to visit http://www.cs.uni-potsdam.de/clasp/.

Optimizations in the Transformation Function

In *DRSys*, we introduced a transformation function that mapped the integrity constraints into ASP. We also devised some optimizations regarding this transformation, such as the projection of the relevant attributes in order to reduce the size of the grounded program. However, by further studying the implementation of the answer set solver, we may perform some new optimizations to the transformation function that, together with *clasp*, increases even more the performance.

Optimizations in the Dependencies Graph

In *DRSys*, we presented a mechanism to automatically determine which relations are relevant to the repair process, with respect to a set of integrity constraints. However, our algorithm is very conservative, i.e., many of the times determines that some relations are relevant when, in reality, they are not. As future work, it would be interesting to create a more "intelligent" algorithm, such that the algorithm could choose more wisely the relevant constraints needed to a specific problem.

Concurrency in Answer Set Solvers

One very interesting idea, would be the use of concurrent programming, by breaking the problem into several smaller problems and dividing them through a set of processors. Then, each of which would compute its corresponding part and in the end, they would unite each part to form the final result. Curiously enough, some steps have already been made towards concurrency in answer set programming. In [SSG+09], there have already been shown some results for some particular classes of problems. Nevertheless, being it an open problem still, it could lead to lots of gain, with respect to the time needed to repair the database.

DRSys is an application built to repair databases whenever the database becomes inconsistent. During this dissertation, we built a very robust an complete system, that allows the user to create, at any point in time, new integrity constraints to be enforced in the database and, if it becomes inconsistent, *DRSys* is able to, by means of insertions and/or deletions, restore consistency.

In this work, we used answer set programming as a tool to solve the database repairing problem. We introduced a transformation function that maps the database repair problem into a logic program. We showed the quality of such mapping by proving its soundness and completeness, with respect to the repairs generated. While mapping the integrity constraints, we focused on the most widely known integrity constraints provided in the database literature: key constraints, functional dependencies, inclusion dependencies and denial constraints. Furthermore, we allow the user to specify all kinds of

integrity constraints that can be mapped into a answer set programming.

When the user defines new integrity constraints, *DRSys* computes the possible repairs and returns the user the operations that are needed to achieve consistency, regarding a particular repair, so that he can choose which repair he wishes to keep. However, *DRSys* does not compute all repairs and present them to the user, since the number of repair can be very high. Therefore, in *DRSys*, we allow the user to choose between two distinct minimality criteria in order to compute repairs. This way, we only present the user with repairs that change as little information as possible.

In order to aid the user throughout the whole process of specifying integrity constraints, we developed a *wizard* based graphical user interface. This way, we provide the user a very intuitive step-by-step interface, such that he can easily interact with *DRSys*.

Since the database repairing problem is very complex, we introduced several interesting additional features in *DRSys* that can greatly increase performance.

We also extensively tested *DRSys* to study how it would scale. Unfortunately, and as expected, since the problem is very complex (recall that the computation complexity of database repairing lies between the *NP-hard* and Σ_2^p complexity classes), *DRSys* does not scale very well. Although the additional features introduced improve performance, it still is a very complex and hard problem. However, one must take into consideration that database repairing is not a technique designed to be used on a daily basis. Therefore, it is not that bad that *DRSys* takes quite some time to perform repairs.

We have also introduced some future work that can be done in this area, in order to increase performance and reduce the time needed to perform optimizations.

To conclude, *DRSys* is a working implementation of the database repair problem, allowing the user to define new integrity constraints and enforce them into the database at any point in time. Although the process is long, the number of times that the repairing process is needed is very limited, making this approach feasible in the real world.

We hope to have motivated the reader to this area of databases, which is still an open problem. We introduced our problem and solution, and, in the end, we pointed several steps that can be taken in order to improve this work. We also hope to have shown how interesting this topic is, and how much more work can be done.

Bibliography

[ABC99] Marcelo Arenas, Leopoldo E. Bertossi, and Jan Chomicki. Consistent query answers in inconsistent databases. In *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 31 - June 2, 1999, Philadelphia, Pennsylvania*, pages 68–79. ACM Press, 1999.

[ABC00] Marcelo Arenas, Leopoldo E. Bertossi, and Jan Chomicki. Specifying and querying database repairs using logic programs with exceptions. In *FQAS*, pages 27–41, 2000.

[ABC03] Marcelo Arenas, Leopoldo E. Bertossi, and Jan Chomicki. Answer sets for consistent query answering in inconsistent databases. *TPLP*, 3(4-5):393–424, 2003.

[ABK00] Marcelo Arenas, Leopoldo E. Bertossi, and Michael Kifer. Applications of annotated predicate calculus to querying inconsistent databases. In John W. Lloyd, Verónica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, *Computational Logic - CL 2000, First International Conference, London, UK, 24-28 July, 2000, Proceedings*, volume 1861 of *Lecture Notes in Computer Science*, pages 926–941. Springer, 2000.

[Bac73] Charles W. Bachman. The programmer as navigator. *Commun. ACM*, 16(11):635–658, 1973.

[BB03] Pablo Barceló and Leopoldo E. Bertossi. Logic programs for querying inconsistent databases. In Verónica Dahl and Philip Wadler, editors, *Practical Aspects of Declarative Languages, 5th International Symposium, PADL 2003, New Orleans, LA, USA, January 13-14, 2003, Proceedings*, volume 2562 of *Lecture Notes in Computer Science*, pages 208–222. Springer, 2003.

[BB06] Loreto Bravo and Leopoldo E. Bertossi. Semantically correct query answers in the presence of null values. In *EDBT Workshops*, pages 336–357, 2006.

[BBB01] Pablo Barceló, Leopoldo E. Bertossi, and Loreto Bravo. Characterizing and computing semantically correct answers from databases with annotated logic and answer sets. In Leopoldo E. Bertossi, Gyula O. H. Katona, Klaus-Dieter Schewe, and Bernhard Thalheim, editors, *Semantics in Databases, Second International Workshop, Dagstuhl Castle, Germany, January 7-12, 2001, Revised Papers*, volume 2582 of *Lecture Notes in Computer Science*, pages 7–33. Springer, 2001.

[BG94] Chitta Baral and Michael Gelfond. Logic programming and knowledge representation. *J. Log. Program.*, 19/20:73–148, 1994.

[BIG10] George Beskales, Ihab F. Ilyas, and Lukasz Golab. Sampling the repairs of functional dependency violations under hard constraints. *PVLDB*, 3(1):197–207, 2010.

[Bry97] François Bry. Query answering in information systems with integrity constraints. In Sushil Jajodia, William List, Graeme W. McGregor, and Leon Strous, editors, *Integrity and Internal Control in Information Systems, IFIP TC11 Working Group 11.5, First Working Conference on Integrity and Internal Control in Information Systems: Increasing the confidence in Information Systems, Zurich, Switzerland, December 4-5, 1997*, volume 109 of *IFIP Conference Proceedings*, pages 113–130. Chapman Hall, 1997.

[BSI+10] George Beskales, Mohamed A. Soliman, Ihab F. Ilyas, Shai Ben-David, and Yubin Kim. Probclean: A probabilistic duplicate detection system. In Feifei Li, Mirella M. Moro, Shahram Ghandeharizadeh, Jayant R. Haritsa, Gerhard Weikum, Michael J. Carey, Fabio Casati, Edward Y. Chang, Ioana Manolescu, Sharad Mehrotra, Umeshwar Dayal, and Vassilis J. Tsotras, editors, *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, pages 1193–1196, 2010.

[BSIBD09] George Beskales, Mohamed A. Soliman, Ihab F. Ilyas, and Shai Ben-David. Modeling and querying possible repairs in duplicate detection. *PVLDB*, 2(1):598–609, 2009.

[Can07] Monica Caniupan. *Optimizing and implementing repair programs for consistent query answering in databases*. PhD thesis, Ottawa, Ont., Canada, Canada, 2007. AAINR23289.

[CB00] Alexander Celle and Leopoldo E. Bertossi. Querying inconsistent databases: Algorithms and implementation. In John W. Lloyd, Verónica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, *Computational Logic*

- *CL 2000, First International Conference, London, UK, 24-28 July, 2000, Proceedings*, volume 1861 of *Lecture Notes in Computer Science*, pages 942–956. Springer, 2000.

[Cho07] Jan Chomicki. Consistent query answering: Five easy pieces. In Thomas Schwentick and Dan Suciu, editors, *Database Theory - ICDT 2007, 11th International Conference, Barcelona, Spain, January 10-12, 2007, Proceedings*, volume 4353 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2007.

[cla] clasp. http://www.cs.uni-potsdam.de/clasp/.

[CM05] Jan Chomicki and Jerzy Marcinkowski. Minimal-change integrity maintenance using tuple deletions. *Inf. Comput.*, 197(1-2):90–121, 2005.

[CM11] Fei Chiang and Renée J. Miller. A unified model for data and constraint repair. In Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan, editors, *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 446–457. IEEE Computer Society, 2011.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[CS98] Jan Chomicki and Gunter Saake, editors. *Logics for Databases and Information Systems (the book grow out of the Dagstuhl Seminar 9529: Role of Logics in Information Systems, 1995)*. Kluwer, 1998.

[CZ06] Luciano Caroprese and Ester Zumpano. A framework for merging, repairing and querying inconsistent databases. In Yannis Manolopoulos, Jaroslav Pokorný, and Timos K. Sellis, editors, *Advances in Databases and Information Systems, 10th East European Conference, ADBIS 2006, Thessaloniki, Greece, September 3-7, 2006, Proceedings*, volume 4152 of *Lecture Notes in Computer Science*, pages 383–398. Springer, 2006.

[DGKS10] Christian Drescher, Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Heuristics in conflict resolution. *CoRR*, abs/1005.1716, 2010.

[dlv] DLV. http://www.dlvsystem.com/dlvsystem/index.php/Home.

[EFLP99] Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. The diagnosis frontend of the dlv system. *AI Commun.*, 12(1-2):99–111, 1999.

[ELM+98] Thomas Eiter, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. The kr system dlv: Progress report, comparisons and benchmarks. In *KR*, pages 406–417, 1998.

[fEotEC98] United Nations. Economic Commission for Europe and Statistical Office of the European Communities. *Recommendations for the 2000 censuses of population and housing in the ECE region*. Statistical standards and studies. United Nations, 1998.

[Fit96] Melvin Fitting. *First-order logic and automated theorem proving (2nd ed.)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[For82] Charles Forgy. Rete: A fast algorithm for the many patterns/many objects match problem. *Artif. Intell.*, 19(1):17–37, 1982.

[FPL+01] Enrico Franconi, Antonio Laureti Palma, Nicola Leone, Simona Perri, and Francesco Scarcello. Census data repair: a challenging application of disjunctive logic programming. In Robert Nieuwenhuis and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 8th International Conference, LPAR 2001, Havana, Cuba, December 3-7, 2001, Proceedings*, volume 2250 of *Lecture Notes in Computer Science*, pages 561–578. Springer, 2001.

[GGZ01] Gianluigi Greco, Sergio Greco, and Ester Zumpano. A logic programming approach to the integration, repairing and querying of inconsistent databases. In Philippe Codognet, editor, *Logic Programming, 17th International Conference, ICLP 2001, Paphos, Cyprus, November 26 - December 1, 2001, Proceedings*, volume 2237 of *Lecture Notes in Computer Science*, pages 348–364. Springer, 2001.

[GGZ03] Gianluigi Greco, Sergio Greco, and Ester Zumpano. A logical framework for querying and repairing inconsistent databases. *IEEE Trans. Knowl. Data Eng.*, 15(6):1389–1408, 2003.

[GKNS07] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. *clasp* : A conflict-driven answer set solver. In Chitta Baral, Gerhard Brewka, and John S. Schlipf, editors, *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007, Tempe, AZ, USA, May 15-17, 2007, Proceedings*, volume 4483 of *Lecture Notes in Computer Science*, pages 260–265. Springer, 2007.

[GL88] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, pages 1070–1080, 1988.

[GL91] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Comput.*, 9(3/4):365–386, 1991.

[GL97] Michael Gertz and Udo W. Lipeck. An extensible framework for repairing constraint violations. In Sushil Jajodia, William List, Graeme W. McGregor,

and Leon Strous, editors, *Integrity and Internal Control in Information Systems, IFIP TC11 Working Group 11.5, First Working Conference on Integrity and Internal Control in Information Systems: Increasing the confidence in Information Systems, Zurich, Switzerland, December 4-5, 1997*, volume 109 of *IFIP Conference Proceedings*, pages 89–111. Chapman Hall, 1997.

[KL09] Solmaz Kolahi and Laks V. S. Lakshmanan. On approximating optimum repairs for functional dependency violations. In Ronald Fagin, editor, *Database Theory - ICDT 2009, 12th International Conference, St. Petersburg, Russia, March 23-25, 2009, Proceedings*, volume 361 of *ACM International Conference Proceeding Series*, pages 53–62. ACM, 2009.

[Len02] Maurizio Lenzerini. Data integration: A theoretical perspective. In Lucian Popa, editor, *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*, pages 233–246. ACM, 2002.

[Lif02] Vladimir Lifschitz. Answer set programming and plan generation. *Artif. Intell.*, 138(1-2):39–54, 2002.

[Lif08] Vladimir Lifschitz. What is answer set programming? In Dieter Fox and Carla P. Gomes, editors, *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, pages 1594–1597. AAAI Press, 2008.

[LLL00] Mong-Li Lee, Tok Wang Ling, and Wai Lup Low. Intelliclean: a knowledge-based intelligent data cleaner. In *KDD*, pages 290–294, 2000.

[LLL01] Wai Lup Low, Mong-Li Lee, and Tok Wang Ling. A knowledge-based approach for duplicate elimination in data cleaning. *Inf. Syst.*, 26(8):585–606, 2001.

[LRS97] Nicola Leone, Pasquale Rullo, and Francesco Scarcello. Disjunctive stable models: Unfounded sets, fixpoint semantics, and computation. *Inf. Comput.*, 135(2):69–112, 1997.

[Mai83] David Maier. *The Theory of Relational Databases*. Computer Science Press, 1983.

[MB05] Mónica Caniupán Marileo and Leopoldo E. Bertossi. Optimizing repair programs for consistent query answering. In *XXV International Conference of the Chilean Computer Science Society, SCCC 2005, 7-11 November 2005, Valdivia, Chile*, pages 3–12. IEEE Computer Society, 2005.

[MB07] Mónica Caniupán Marileo and Leopoldo E. Bertossi. The consistency extractor system: Querying inconsistent databases using answer set programs. In

Henri Prade and V. S. Subrahmanian, editors, *Scalable Uncertainty Management, First International Conference, SUM 2007, Washington, DC, USA, October 10-12, 2007, Proceedings*, volume 4772 of *Lecture Notes in Computer Science*, pages 74–88. Springer, 2007.

[MS89] V. Wiktor Marek and V. S. Subrahmanian. The relationship between logic program semantics and non-monotonic reasoning. In *ICLP*, pages 600–617, 1989.

[MT98] Victor W. Marek and Miroslaw Truszczynski. Stable models and an alternative logic programming paradigm. *CoRR*, cs.LO/9809032, 1998.

[NS97] Ilkka Niemelä and Patrik Simons. Smodels - an implementation of the stable model and well-founded semantics for normal lp. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *Logic Programming and Nonmonotonic Reasoning, 4th International Conference, LPNMR'97, Dagstuhl Castle, Germany, July 28-31, 1997, Proceedings*, volume 1265 of *Lecture Notes in Computer Science*, pages 421–430. Springer, 1997.

[Prz91] Teodor C. Przymusinski. Stable semantics for disjunctive programs. *New Generation Comput.*, 9(3/4):401–424, 1991.

[RD00] Erhard Rahm and Hong Hai Do. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.*, 23(4):3–13, 2000.

[Rei82] Raymond Reiter. Towards a logical reconstruction of relational database theory. In *On Conceptual Modelling (Intervale)*, pages 191–233, 1982.

[RG03] Raghu Ramakrishnan and Johannes Gehrke. *Database management systems (3. ed.)*. McGraw-Hill, 2003.

[SKS05] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts, 5th Edition*. McGraw-Hill Book Company, 2005.

[SMG10] Emanuel Santos, João Pavão Martins, and Helena Galhardas. An argumentation-based approach to database repair. In Helder Coelho, Rudi Studer, and Michael Wooldridge, editors, *ECAI 2010 - 19th European Conference on Artificial Intelligence, Lisbon, Portugal, August 16-20, 2010, Proceedings*, volume 215 of *Frontiers in Artificial Intelligence and Applications*, pages 125–130. IOS Press, 2010.

[smo] smodels. `http://www.tcs.hut.fi/Software/smodels/`.

[SSG+09] Lars Schneidenbach, Bettina Schnor, Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Experiences running a parallel answer set solver on blue gene. In *PVM/MPI*, pages 64–72, 2009.

[VS09] Navin Viswanath and Rajshekhar Sunderraman. Source-aware repairs for inconsistent databases. In Qiming Chen, Alfredo Cuzzocrea, Takahiro Hara, Ela Hunt, and Manuela Popescu, editors, *The First International Conference on Advances in Databases, Knowledge, and Data Applications, DBKDS 2009, Gosier, Guadeloupe, France, 1-6 March 2009*, pages 125–130. IEEE Computer Society, 2009.

[Wij03] Jef Wijsen. Condensed representation of database repairs for consistent query answering. In Diego Calvanese, Maurizio Lenzerini, and Rajeev Motwani, editors, *Database Theory - ICDT 2003, 9th International Conference, Siena, Italy, January 8-10, 2003, Proceedings*, volume 2572 of *Lecture Notes in Computer Science*, pages 375–390. Springer, 2003.

[Wij05] Jef Wijsen. Database repairing using updates. *ACM Trans. Database Syst.*, 30(3):722–768, 2005.

[xsb] XSB. http://xsb.sourceforge.net/.

[ZMMM01] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD*, pages 279–285, 2001.